UNIVERSITÀ DI PADOVA FACOLTÀ DI INGEGNERIA

Introduzione a Ruby

Tesina

Laureando: Simone Ranzato

Relatore: Prof. Federico Filira

Corso di laurea in
Ingegneria Informatica

Anno Accademico 2009 – 2010

Indice

	Indice	1
	Introduzione	3
1.	Caratteristiche generali di Ruby	5
2.	Guida rapida	7
	0. Come iniziare	8
	1. I numeri	10
	2. Le lettere	12
	3. Variabili e assegnazioni	15
	4. Mescoliamoli insieme	17
	5. Qualcos'altro sui metodi	21
	6. Controllo del flusso	27
	7. Array e iteratori	34
	8. Scrivere metodi propri	37
	9. Le classi	50
	10. Blocchi e procedure	59
3.	Uso di Ruby per creare una documentazione html	67
4.	Bibliografia	77

Introduzione

Obiettivo di questa tesina è realizzare una guida rapida su Ruby. Per la realizzazione di questa si parte dal testo *Learn to program* di Chris Pine (si può trovare una versione online a questo indirizzo: http://pine.fm/LearnToProgram/). Il testo risulta essere molto semplice e scorrevole, e sebbene non sia una guida completa alla programmazione in Ruby, va comunque a trattare tutti i punti principali necessari ad un primo approccio a questo linguaggio.

Il passo successivo è stato quello di cercare di creare della documentazione in html da un file txt, utilizzando opportune convenzioni (io utilizzerò la suddetta guida realizzata), utilizzando del codice scritto in Ruby.

1. Caratteristiche generali di Ruby

Ruby è un linguaggio di scripting gratuito e completamente ad oggetti. Nasce come progetto personale del giapponese Yukihiro Matsumoto, chiamato anche semplicemente “Matz”, che ha iniziato a sviluppare il linguaggio nel 1993 (la prima release è del 1995).

Matz ha fuso insieme parti dei suoi linguaggi di programmazione preferiti (Perl, Smalltalk, Eiffel, Ada e Lisp) per creare un nuovo linguaggio in grado di bilanciare programmazione funzionale con programmazione imperativa.

> *I believe people want to express themselves when they program.*
> *They don’t want to fight with the language.*
> *Programming languages must feel natural to programmers.*
>
> *Matz*

Negli ultimi anni la popolarità di Ruby ha avuto ancora maggior successo soprattutto grazie alla comparsa di framework di successo per lo sviluppo di applicazioni web come Ruby On Rails.

In Ruby, ogni cosa è un oggetto: ogni parte di informazione e codice ha delle sue proprietà e azioni. La programmazione ad oggetti chiama le proprietà con il nome *variabili di istanza* e le azioni sono conosciute come *metodi*. L’approccio puramente orientato agli oggetti di Ruby è facilmente dimostrabile dalla seguente porzione di codice che applica un’azione ad un numero:

```
5.times { puts "Ruby è un linguaggio fantastico" }
```

che produce il seguente output:

```
Ruby e' un linguaggio fantastico

Ruby e' un linguaggio fantastico
```

```
Ruby e' un linguaggio fantastico
Ruby e' un linguaggio fantastico
Ruby e' un linguaggio fantastico
```

In molti linguaggi, i numeri e gli altri tipi primitivi non sono oggetti, ma Ruby, seguendo l'esempio di Smalltalk, dà metodi e variabili di istanza a tutti i suoi tipi. Questo rende più facile l'utilizzo di Ruby, dal momento che tutte le regole applicabili agli oggetti si applicano, di fatto, all'intero linguaggio.

Si tratta di un linguaggio interpretato, cioè scritto in puro testo e tradotto ad ogni esecuzione da un programma chiamato interprete, e non è necessaria una fase preventiva di compilazione. Questo permette una maggiore agilità del codice, in particolar modo per operazioni di debugging, anche se a scapito della sua efficienza.

Ruby è un linguaggio multipiattaforma e può essere utilizzato tranquillamente sia sui principali sistemi operativi (Windows, Linux e Mac OS) che su quelli meno conosciuti.

Il testo utilizzato come base riferimento alla versione 1.8.4 di Ruby; attualmente si è giunti alla release 1.9.1 del programma (il changelog della versione 1.9.1 lo si può trovare qui: http://svn.ruby-lang.org/repos/ruby/tags/v1_9_1_0/NEWS), ma per le caratteristiche base, a livello di codice, non vi sono grandi differenze.
La versione 1.9.1 è una nuova versione di Ruby moderna, più veloce, con sintassi più chiara e con supporto multilingua. Uno dei grandi difetti delle versioni precedenti di Ruby era la lentezza. Per questo motivo, a partire da questa versione, Ruby integra YARV (*Yet another Ruby VM)*, una virtual machine per l'interprete che lavora producendo un byte code del file sorgente ed interpretandolo; questa nuova tecnica unita ad altri miglioramenti ha fatto si che i tempi di interpretazione siano notevolmente ridotti, rendendo più performante l'esecuzione.

Ruby si può scaricare nel sito http://www.ruby-lang.org/it/downloads/ oppure su http://rubyinstaller.org/download.html, dove si possono trovare sia gli eseguibili che i file compilati. Sarà poi necessario solo un editor (ad esempio Notepad++, SciTE, ecc) che ci aiuti nella scrittura dei programmi.

2. Guida rapida

Viene riportata di seguito la realizzazione di una semplice *guida rapida* su Ruby, che va a toccare un po' tutto ciò di cui si ha bisogno per un primo approccio a questo linguaggio di programmazione. La guida è stata realizzata facendo riferimento al libro *Learn to program* di Chris Pine.

Tale guida è servita ad analizzare le caratteristiche base di Ruby, al fine di arrivare ad analizzare il codice Ruby per generare la documentazione html.

Questa stessa guida verrà poi utilizzata come file di partenza, in formato txt, per la conversione attraverso codice Ruby in file html.

Impariamo a programmare

0. Come iniziare

Quando si programma un computer, è necessario “parlare” in un linguaggio che il computer capisca: un linguaggio di programmazione. Ci sono decine e decine di linguaggi differenti, e molti di questi sono eccellenti. In questa guida viene trattato il linguaggio di programmazione Ruby.

Quando scriviamo qualcosa nel linguaggio umano, ciò che viene scritto lo chiamiamo testo. Quando scriviamo qualcosa in un linguaggio di programmazione, questo viene invece chiamato *codice*. Ho inserito molti esempi di codice Ruby nella guida, la maggior parte dei quali sono programmi completi che potete eseguire sul vostro computer. Per rendere la lettura del codice più immediata ho colorato differentemente le varie parti (per esempio, i numeri sono sempre in **`verde`**). Tutto ciò che si dovrà digitare sarà in un `rettangolo bianco` e tutto ciò che il programma stamperà sarà in un `rettangolo blu`.

Ma innanzitutto dobbiamo scaricare e installare Ruby sul nostro computer!

Installazione per Windows

L’installazione di Ruby per Windows è velocissima. Come prima cosa bisogna scaricare il programma di installazione di Ruby. Potrebbero esserci più di una versione tra le quali scegliere; questa guida usa la versione 1.8.4, quindi assicuriamoci di scaricare una versione recente almeno quanto questa (io suggerirei di scaricare l’ultima versione disponibile, attualmente la 1.9.1). Dopodiché semplicemente eseguiamo il programma di installazione. Ci verrà chiesto dove installare Ruby e almeno che non si abbia una buona ragione per farlo, si lasci la sua locazione di default.

Per poter programmare dobbiamo poter scrivere programmi ed eseguirli. Per fare questo c’è bisogno di un editor di testi e di una linea di comando.

Il programma di installazione di Ruby installa anche un bel editor di testi chiamato SciTE (Scintilla Text Editor). Per far colorare il proprio codice alla stessa maniera degli esempi di questa guida, scarichiamo i seguenti file e copiamoli nella cartella di SciTE (`c:/ruby/scite` se non è stata modificata la sua locazione):

- Proprietà globali (http://pine.fm/LearnToProgram/SciTEGlobal.properties)
- Proprietà di ruby (http://pine.fm/LearnToProgram/ruby.properties)

Sarebbe inoltre una buona idea quella di creare una cartella dove conservare tutti i propri programmi. Assicuriamoci quando salviamo un programma di farlo in questa cartella.

Per ottenere la linea di comando, selezioniamo “Prompt dei comandi” dalla cartella Accessori del menu avvio. Quindi spostiamoci nella cartella dove conserviamo i nostri programmi.

Digitando `cd..` si scenderà di un livello e digitando `cd nomecartella` si entrerà nella cartella con nome `nomecartella`. Per vedere tutte le cartelle contenute nella cartella in cui ci si trova, digitiamo `dir /ad`.

Ed è tutto! È tutto pronto per imparare a programmare.

Installazione per Macintosh

Se utilizziamo Mac OS X 10.2 (Jaguar), allora abbiamo già Ruby a disposizione sul nostro sistema! Potrebbe essere più facile di così? Se invece stiamo utilizzando la versione 10.1, avremo bisogno di scaricare Ruby. Sfortunatamente credo che non sia possibile utilizzare Ruby su Mac OS 9 e sulle versioni precedenti.

Per poter programmare abbiamo bisogno di poter scrivere programmi ed eseguirli. Per fare questo c'è bisogno di un editor di testi e di una linea di comando.

La linea di comando è accessibile tramite l'applicazione Terminal (che si trova in Applications/Utilities).

Come editor di testi possiamo utilizzare qualunque editor al quale si è familiari o con i quali ci si trova bene. Se si utilizza TextEdit, comunque, assicuriamoci di salvare i programmi nel formato di solo testo altrimenti i programmi *non funzioneranno*! Altri editor sono emacs, vi e pico, tutti accessibili dalla linea di comando.

Ed è tutto! È tutto pronto per imparare a programmare.

Installazione per Linux

Innanzitutto controlliamo se Ruby è già installato sul nostro sitema. Digitiamo `which ruby`. Se viene stampato qualcosa tipo `/usr/bin/which: no ruby in (...)`, allora avremo bisogno di scaricare Ruby, altrimenti controlliamo che versione di Ruby è installata digitando `ruby -v`. Se è più vecchia dell'ultima versione stabile rilasciata, possiamo aggiornarla dalla pagina suddetta.

Se si abbiamo i permessi di root (amministratore), allora probabilmente non abbiamo necessità di conoscere alcuna istruzione per installare Ruby. Se invece non li abbiamo, dovremo chiedere all'amministratore di sistema di installarlo (in questo modo Ruby potrà essere utilizzato da tutti gli utenti del sistema).

Altrimenti potremo installarlo in modo tale che solo il nostro utente potrà utilizzarlo. Spostiamo il file scaricato in una directory temporanea, tipo `$HOME/tmp`. Se il nome del file è `ruby-1.8.7-p72.tar.gz`, è possibile estrarlo digitando `tar zxvf ruby-1.8.7-p72.tar.gz`. Spostiamoci nella directory appena creata (in questo esempio, `cd ruby-1.8.7-p72`).

Configurare l'installazione digitando `./configure --prefix=$HOME`. Quindi scriviamo `make` per assemblare il nostro interprete Ruby. Questa operazione potrebbe richiedere qualche minuto; al termine, digitiamo `make install` per installarlo.

Infine avremo bisogno di aggiungere `$HOME/bin` al nostro path modificando il file `$HOME/.bashrc` (Perché questa modifica abbia effetto bisogna fare il logout e nuovamente il login). Fatto questo, controlliamo che l'installazione sia andata a buon termine digitando: `ruby -v`. Se il programma risponde con la versione di Ruby installata è possibile eliminare i file nella directory `$HOME/tmp` (o dovunque siano stati messi).

Ed è tutto! È tutto pronto per imparare a programmare.

1. I numeri

Ora che sappiamo tutto sulla configurazione siamo pronti per scrivere un programma! Apriamo il nostro editor preferito e scriviamo la seguente riga:

```
puts 1+2
```

Salviamo il nostro programma (si, è un programma!) come `calc.rb` (**.rb** è la tipica estensione dei programmi scritti in Ruby). Adesso eseguiamo il nostro programma scrivendo `ruby calc.rb` in un terminale. Dovrebbe apparire `3` sul nostro schermo. Visto, programmare non è poi così difficile, non è così?

Introduzione a `puts`

`puts` scrive semplicemente sullo schermo quello che viene dopo di lui.

```
puts 3
```

```
3
```

Interi e decimali

In molti linguaggi di programmazione (e Ruby non fa eccezione) i numeri senza virgola sono chiamati *interi*, e quelli con la virgola sono in genere chiamati *decimali* o *numeri a virgola mobile*.

Ecco alcuni interi:

```
5
-205
9999999999999999999999999
0
```

E questi sono alcuni decimali:

```
54.321
0.001
-205.3884
0.0
```

In pratica, molti programmi non usano i decimali, ma solo numeri interi. (Dopo tutto, nessuno vuole leggere 7.4 email, o navigare 1.8 pagine, oppure ascoltare 5.24 delle proprie canzoni preferite...). I decimali sono usati in ambiti accademici (esperimenti di fisica e simili) e per la grafica 3D. Anche molti programmi che hanno a che fare con i soldi usano interi, tengono semplicemente il conto del numero dei centesimi!

Semplice Aritmetica

Finora abbiamo ottenuto tutte le funzionalità di una semplice calcolatrice. (Le calcolatrici usano sempre i decimali, quindi se vogliamo che il nostro computer si comporti come una calcolatrice dobbiamo usare anche noi i decimali). Per l'addizione e la sottrazione usiamo + e – come già detto. Per la moltiplicazione usiamo `*` e per la divisione `/`. Molte tastiere hanno questi tasti nel tastierino numerico a destra.

Proviamo ora ad aggiungere qualcosa al nostro programma calc.rb. Scriviamo quanto segue ed eseguiamo.

```
puts 1.0 + 2.0
puts 2.0 * 3.0
puts 5.0 - 8.0
puts 9.0 / 2.0
```

Otterremo questo:

```
3.0
6.0
-3.0
4.5
```

(Gli spazi nel programma non sono importanti; rendono solo il codice facile da leggere.) Bene, finora nessuna grande sorpresa. Proviamo adesso con gli interi:

```
puts 1+2
puts 2*3
puts 5-8
puts 9/2
```

Essenzialmente gli stessi di prima, giusto?

```
3
6
-3
4
```

Uh... tranne l'ultimo! Quando si fa aritmetica con gli interi bisogna aspettarsi risposte *intere*. Quando il computer non può ricavare la risposta *esatta* non può far altro che arrotondare.

(Ovviamente, `4` è la risposta giusta di `9/2` nell'aritmetica degli interi; solo non è la risposta che ci aspettavamo.)

Forse vi starete chiedendo a cosa serve la divisione tra numeri interi. Bene, immaginiamo di andare a vedere un film e di avere solo 9 dollari. Qui a Padova si può vedere un film al Lux con 2 euro. Quanti film possiamo vedere? `9/2`... `4` film. In questo caso di sicuro `4.5` *non* è la risposta giusta: non ci verrà fatto vedere mezzo film e non si potrà entrare solo per metà per vedere un film intero... alcune cose semplicemente sono indivisibili.

Sperimentate adesso per conto vostro con qualche programma! Per scrivere espressioni più complesse si possono usare le parentesi. Ad esempio:

```
puts 5 * (12-8) + -15
puts 98 + (59872 / (13*8)) * -52
```

```
5
-29802
```

Alcune cose da provare

Scrivere un programma che dica:

- quante ore ci sono in un anno?
- quanti minuti ci sono in un decennio?
- quanti secondi di vita hai?
- quanti cioccolatini conti di mangiare nella tua vita?
  Attenzione: questa parte del programma potrebbe richiedere un po' di tempo!

Ecco una domanda più ostica:

- Se ho 959 milioni di secondi di vita, quanti anni ho?

Una volta finito di giocare e sperimentare con i numeri, andremo a dare un'occhiata alle lettere.

2. Le lettere

E così adesso sappiamo tutto sui numeri, ma cosa sappiamo sulle lettere? e sulle parole? e sul testo?

In un programma un gruppo di lettere è detto *stringa*. (Potete pensare a delle lettere stampate tenute insieme su uno striscione). Le stringhe saranno colorate di `rosso` per distinguerle facilmente all'interno del codice. Ecco alcune stringhe:

```
'Ciao.'
'Ruby rocks.'
'5 è il mio numero preferito... il tuo?'
```

```
'Snoopy dice #%^?&@! quando urta con il ditone.'
'   '
''
```

Si può notare che le stringhe possono contenere punteggiatura, cifre, simboli... oltre alle lettere. L'ultima stringa non contiene niente; perciò la chiameremo *stringa vuota*.

Prima abbiamo usato `puts` per stampare dei numeri; adesso proveremo con alcune stringhe:

```
puts 'Ciao Mondo!'
puts ''
puts 'Addio.'
```

```
Ciao Mondo!

Addio.
```

Funziona bene. Adesso prova con alcune stringhe a tuo piacere.

Aritmetica con le stringhe

Così come si può fare aritmetica con i numeri, è possibile fare aritmetica con le stringhe! O almeno una specie di aritmetica... comunque possiamo sommare le stringhe. Proviamo a sommare due stringhe e vediamo puts cosa tira fuori.

```
puts 'Mi piace' + 'la torta di mele.'
```

```
Mi piacela torta di mele.
```

Oops! Ho dimenticato di mettere uno spazio tra **`'Mi piace'`** e **`'la torta di mele'`**. Gli spazi in genere non contano, ma hanno importanza all'interno delle stringhe. (È vero quello che si dice: i computer non fanno quello che *vorremmo* che loro facessero, ma fanno solo quello che gli si *dice* di fare.) Riproviamo ancora:

```
puts 'Mi piace ' + 'la torta di mele.'
puts 'Mi piace' + ' la torta di mele.'
```

```
Mi piace la torta di mele.
Mi piace la torta di mele.
```

(Come si può vedere non importa in quale stringa è stato aggiunto lo spazio.)

Quindi possiamo aggiungere stringhe, e possiamo anche moltiplicarle! (Per un numero, ovviamente.) Vediamo:

```
puts 'bau ' * 4
```

```
bau bau bau bau
```

Pensandoci ha perfettamente senso. Dopotutto, `7*3` significa `7+7+7`, così `'moo'*3` significa `'moo'+'moo'+'moo'`.

`12` vs `'12'`

Prima di andare avanti dovremmo essere sicuri di aver capito la differenza tra *numeri* e *cifre*. `12` è un numero, ma `'12'` è una stringa di due cifre.

Giochiamoci sopra per qualche momento:

```
puts 12 + 12
puts '12' + '12'
puts '12 + 12'
```

```
24
1212
12 + 12
```

Qualcosa come questo:

```
puts 2 * 5
puts '2' * 5
puts '2 * 5'
```

```
10
22222
2 * 5
```

Problemi

A questo punto potreste aver provato alcune cose che *non funzionano*. Se così non è stato eccone alcune:

```
puts '12' + 12
puts '2' * '5'
```

```
#<TypeError: can't convert Fixnum into String>
```

Hmmm... un messaggio d'errore. Il problema è che non si può aggiungere un numero ad una stringa, o moltiplicare una stringa per un'altra stringa. Non ha alcun senso più di quanto fare qualcosa del genere:

```
puts 'Betty' + 12
puts 'Fred' * 'John'
```

Qualcos'altro a cui stare attenti: si può scrivere `'maiale'*5` in un programma, che significa `5` ripetizioni della stringa `'maiale'`. Tuttavia *non è possibile* scrivere `5*'maiale'`, che significherebbe `'maiale'` ripetizioni del numero `5`, che è semplicemente stupido.

Infine, proviamo a fare un programma che stampi `You're swell!`:

```
puts 'You're swell!'
```

Ebbene, *così* non funziona. Ecco perché è una buona cosa avere un editor di testo con la *sintassi colorata*. Quindi come facciamo a far sapere al computer cosa vogliamo mettere nella stringa? Dobbiamo fare l'*escape* dell'apostrofo, in questo modo:

```
puts 'You\'re swell!'
```

```
You're swell!
```

Il backslash è il carattere di escape. In altre parole se si ha un backslash seguito da un altro carattere questi a volte vengono sostituiti da un nuovo carattere. Le sole cose di cui il backslash fa l'escape comunque sono l'apostrofo e il backslash stesso (se ci si pensa, i caratteri di escape devono sempre fare l'escape anche di sé stessi). Ecco degli esempi, credo, ordinati:

```
puts 'L\'arcobaleno e\' bello!'
puts 'backslash alla fine della stringa: \\'
puts 'su\\giu'
puts 'su\giu'
```

```
L'arcobaleno e' bello!
backslash alla fine della stringa: \
su\giu
su\giu
```

Dato che il backslash *non* fa l'escape della lettera `'g'`, ma *fa* l'escape di sé stesso, le ultime due stringhe sono identiche: nel codice sembrano differenti ma per il computer sono la stessa stringa.

3. Variabili e assegnazioni

Finora ogni volta che abbiamo usato **`puts`** con una stringa o con un numero, ciò che abbiamo stampato è poi scomparso. Quello che intendo dire è che se volevamo stampare qualcosa due volte, avremmo dovuto scriverla due volte:

```
puts '...puoi dirlo ancora...'
puts '...puoi dirlo ancora...'
```

```
...puoi dirlo ancora...
...puoi dirlo ancora...
```

Sarebbe carino poterlo digitare una sola volta e poi... memorizzarlo da qualche parte. Naturalmente possiamo farlo.

Per memorizzare la stringa nella memoria del nostro computer, bisogna dare un nome alla stringa. I programmatori spesso chiamano questa operazione *assegnazione* e chiamano i nomi *variabili*. La variabile può essere costituita da una sequenza di lettere e numeri, ma il primo carattere deve essere una lettera minuscola. Proviamo nuovamente questo programma, questa

volta però daremo alla stringa il nome **myString** (anche se l'avrei potuta chiamare **str** o **laMiaStringa** oppure **enricoOttavo**).

```
myString = '...puoi dirlo ancora...'
puts myString
puts myString
```

```
...puoi dirlo ancora...
...puoi dirlo ancora...
```

Ogni volta che il programma fa un'operazione con **myString**, in realtà la fa con **'...puoi dirlo ancora...'**. Si può pensare che la variabile **myString** "punti" alla stringa **'...puoi dirlo ancora...'**. Ecco un esempio un po' più interessante:

```
nome = 'Patricia Rosanna Jessica Mildred Oppenheimer'
puts 'Mi chiamo ' + nome + '.'
puts 'Wow! ' + nome + ' e\' un nome veramente lungo!'
```

```
Mi chiamo Patricia Rosanna Jessica Mildred Oppenheimer.
Wow! Patricia Rosanna Jessica Mildred Oppenheimer e' un nome  veramente
lungo!
```

Inoltre, allo stesso modo di come si assegna un oggetto ad una variabile, è possibile *riassegnare* un oggetto differente a quella variabile (ecco perché le chiamiamo variabili: perché ciò a cui puntano può variare).

```
compositore = 'Mozart'
puts compositore + ' era "il massimo" ai suoi tempi.'

compositore = 'Beethoven'
puts 'Ma personalmente preferisco ' + compositore + '.'
```

```
Mozart era "il massimo" ai suoi tempi.
Ma personalmente preferisco Beethoven.
```

Naturalmente le variabili possono puntare a ogni tipo di oggetto, non solo alle stringhe:

```
var = 'solo un\'altra ' + 'stringa'
puts var

var = 5 * (1+2)
puts var
```

```
solo un'altra stringa
15
```

Infatti le variabili possono puntare a quasi qualsiasi cosa eccetto altre variabili. Allora cosa succede se ci proviamo?

```
var1 = 8
var2 = var1
puts var1
```

```
puts var2

puts ''

var1 = 'otto'
puts var1
puts var2
```

```
8
8

otto
8
```

Come prima cosa, quando abbiamo provato a far puntare **`var2`** a **`var1`**, la variabile ha davvero puntato a **`8`** (proprio come **`var1`**). Dopodiché abbiamo fatto puntare **`var1`** a **`'otto'`**, ma dato che **`var2`** in realtà non ha mai veramente puntato **`var1`**, la variabile ha continuato a puntare **`8`**.

Ora che abbiamo le variabili, i numeri e le stringhe, impariamo ad usarli assieme!

4. Mescoliamoli insieme

Finora abbiamo visto alcuni differenti tipi di oggetti (numeri e lettere), e abbiamo creato delle variabili che puntano ad essi; la prossima cosa che faremo sarà quella di prenderli e di metterli insieme.

Abbiamo visto che se vogliamo far stampare **`25`** al nostro programma, il seguente programma ***non** va bene*, poiché non si possono sommare numeri e stringhe:

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Parte del problema sta nel fatto che il computer non sa se vogliamo ottenere **`7`** (**`2 + 5`**) oppure **`25`** (**`'2' + '5'`**).

Prima di poterli sommare abbiamo bisogno di rappresentare **`var1`** come stringa oppure **`var2`** come intero.

Conversioni

Per ottenere la versione stringa di un oggetto, basta aggiungere **`.to_s`** dopo di esso:

```
var1 = 2
var2 = '5'
puts var1.to_s + var2
```

```
25
```

Allo stesso modo, **to_i** ci fornisce l'oggetto sotto forma di intero, e **to_f** ci dà invece la versione decimale. Vediamo un po' più da vicino cosa fanno (e cosa *non fanno*) questi tre metodi:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
puts var1 + var2.to_i
```

```
25
7
```

Si può notare che anche dopo aver convertito **var1** in stringa chiamando **to_s**, questi punta sempre a **2** e mai a **'2'**. A meno che non sia esplicitamente riassegnato **var1** (il che richiede il segno =), questo punterà a **2** per tutta la durata del programma.

Adesso proviamo alcune interessanti (a volte un po' strane) conversioni:

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts ''
puts '5 is my favorite number!'.to_i
puts 'Who asked you about 5 or whatever?'.to_i
puts 'Your momma did.'.to_f
puts ''
puts 'stringy'.to_s
puts 3.to_i
```

```
15.0
99.999
99

5
0
0.0

stringy
3
```

Probabilmente, alcune di queste conversioni sorprendono un po'. La prima è abbastanza standard e ci da **15.0**. Poi abbiamo convertito la stringa **'99.999'** in decimale e in intero. Il decimale fa quello che ci si aspetta; l'intero, come al solito, è arrotondato.

E ancora, seguono alcuni esempi di... *insolite* stringhe convertite in numeri. **to_i** ignora la prima cosa che non capisce, e anche il resto della stringa da quel punto in poi. Così la prima stringa è convertita in **5**, ma le altre, che iniziano con una lettera, sono ignorate completamente... e il computer mostra solo uno zero.

Infine, le ultime due conversioni non fanno assolutamente niente, proprio come ci si aspetta.

Un altro sguardo a `puts`

Diamo uno sguardo a questo:

```
puts 20
puts 20.to_s
puts '20'
```

```
20
20
20
```

Perché in tutti e tre i casi viene stampata la stessa cosa? Bene, sugli ultimi due nessuna stranezza, poiché **`20.to_s`** è **`'20'`**. Ma per quanto riguarda il primo, il `20` intero? A questo proposito, che cosa significa scrivere *l'intero* `20`? Quando scriviamo un *2* e poi uno *0* su un pezzo di carta, noi stiamo scrivendo una stringa, non un intero. *L'intero 20* è il numero di dita delle mani e dei piedi che io ho; non è un *2* seguito da uno *0*.

Bene, qui c'è il grande segreto di **`puts`**: prima di provare a scrivere un oggetto **`puts`** utilizza **`to_s`** per ottenere l'oggetto sotto forma di stringa. Infatti, la *s* in **`puts`** sta per *stringa*; **`puts`** dunque significa *put string*.

Questo potrebbe sembrare non molto entusiasmante adesso, ma in Ruby ci sono tantissimi tipi di oggetti (impareremo anche come crearne di nostri!), ed è divertente sapere cosa accade quando tentiamo di usare **`puts`** su un oggetto veramente molto strano, come ad esempio un'immagine di nostra nonna, o un file musicale o qualcos'altro. Ma questo lo vedremo dopo...

Nel frattempo, vediamo alcuni metodi che ci permetteranno di scrivere ogni genere di programma...

I metodi `gets` e `chomp`

Se **`puts`** significa *stampa*, sono sicuro che potete indovinare il significato di **`gets`**. Poiché **`puts`** tira fuori delle stringhe, allora **`gets`** sarà in grado di recuperare delle stringhe. E da dove le otterrà?

Da noi! Dalla nostra tastiera ovviamente. Poiché la nostra tastiera produce solo stringhe, tutto funziona a meraviglia. Quello che accade è che **`gets`** se ne sta in attesa di leggere quello che scriviamo fino a che non premiamo il tasto `Enter`. Proviamo:

```
puts gets
```

```
Ciao!
Ciao!
```

Ovviamente qualsiasi cosa scriviamo sarà ripetuta. Provate ad eseguirlo più volte e a scrivere cose differenti.

A questo punto siamo pronti per fare un programma interattivo! Scriveremo il nostro nome e il programma ci saluterà:

```
puts 'Ciao, come ti chiami?'
nome = gets
puts 'Ti chiami ' + nome + '? Che bel nome!'
puts 'Piacere di conoscerti, ' + nome + '. :)'
```

Ecco, lo eseguiamo, scriviamo il nostro nome e questo è quello che accade:

```
Ciao, come ti chiami?
Chris
Ti chiami Chris
? Che bel nome!
Piacere di conoscerti, Chris
. :)
```

Hmmm... sembra come se quando ho scritto le lettere `C`, `h`, `r`, `i`, `s`, e premuto `Enter`, **`gets`** ha preso tutte le lettere del mio nome *e* anche `Enter`! Fortunatamente, c'è un metodo proprio per questo tipo di cose: **`chomp`** che toglie ogni `Enter` dalla fine delle nostre stringhe. Proviamo ancora, ma stavolta ci faremo aiutare da **`chomp`**:

```
puts 'Ciao, come ti chiami?'
nome = gets.chomp
puts 'Ti chiami ' + nome + '? Che bel nome!'
puts 'Piacere di conoscerti, ' + nome + '. :)'
```

```
Ciao, come ti chiami?
Chris
Ti chiami Chris? Che bel nome!
Piacere di conoscerti, Chris. :)
```

Molto meglio! Notate che **`nome`** sta puntando a **`gets.chomp`**, quindi non dovremo più fare **`nome.chomp`**, poiché **`nome`** è già **`chomp`**ato.

Alcune cose da provare

- Scrivere un programma che chiede il nome, il secondo nome e il cognome di una persona. E infine dovrebbe salutare la persona usando il nome completo.
- Scrivere un programma che chiede ad una persona il suo numero preferito. Quindi il programma aggiunge uno al numero e lo suggerisce come un *migliore e più grande* numero preferito. (Abbiate molto tatto a tale proposito.)

5. Qualcos'altro sui metodi

Fin qui abbiamo visto alcuni metodi (**puts**, **gets**, **chomp**, **to_i**, **to_f**, **to_s**, **+**, **-**, ***** e **/**), ma non abbiamo mai veramente parlato di cosa sono i metodi. Sappiamo cosa fanno, ma non sappiamo cosa sono.

Ecco cosa sono effettivamente: cose che fanno cose. Se gli oggetti (come stringhe, interi, e decimali) sono i nomi nel linguaggio Ruby, allora i metodi sono come i verbi. E, proprio come nella lingua italiana, non si possono avere verbi senza un nome che li *esegue*. Ad esempio, il ticchettio non è qualcosa che semplicemente accade; un orologio (o una sveglia o qualcosa di simile) lo produce. In italiano diremmo, "L'orologio ticchetta". In Ruby potremmo scrivere **orologio.ticchetta** (ovviamente assumendo che orologio è un oggetto Ruby). I programmatori potrebbero dire che "abbiamo chiamato il metodo **ticchetta** di **orologio**", oppure che "chiamiamo **ticchetta** su **orologio**".

Come dicevo, così come ogni verbo ha bisogno di un sostantivo, allo stesso modo ogni metodo ha bisogno di un oggetto. Solitamente è semplice dire quale oggetto sta eseguendo il metodo: è quello che viene prima del punto, come nel nostro esempio **orologio.ticchetta**, oppure **101.to_s**. A volte, comunque, non è così scontato; come succede per i metodi aritmetici. Risulta che **5 + 5** è solo un modo breve di scrivere **5.+ 5**. Ad esempio:

```
puts 'hello '.+ 'world'
puts (10.* 9).+ 9
```

```
hello world
99
```

Non è molto gradevole, quindi non scriveremo mai in questo modo; comunque, l'importante è capire cosa è successo *veramente*. (Sulla mia macchina, che restituisce anche un *warning*: **warning: parenthesize argument(s) for future version**. Il codice viene eseguito correttamente, ma vengo avvertito che sta avendo problemi a capire cosa voglio dire, e mi invita ad usare più parentesi in futuro.) Questo ci permette di capire anche perché possiamo scrivere **'pig'*5** ma non possiamo scrivere **5*'pig'**: **'pig'*5** sta dicendo a **'pig'** di moltiplicarsi, mentre **5*'pig'** sta dicendo a **5** di moltiplicarsi. **'pig'** sa come fare **5** copie di se stesso e sommarle tutte insieme; invece **5** potrebbe avere molta più difficoltà a fare **'pig'** copie di se stesso e a sommarle insieme.

E, naturalmente, ci sono **puts** e **gets** da spiegare. Dove sono i loro oggetti? In italiano, si può qualche volta sottintendere il nome; ad esempio, se un furfante urla "Crepa!", il nome implicito è colui a cui sta urlando contro. In Ruby, se dico **puts 'essere o non essere'**, in realtà sto dicendo **self.puts 'essere o non essere'**. Dunque cos'è **self**? È una variabile speciale che punta all'oggetto che la contiene. Non sappiamo ancora cosa vuol dire *essere contenuto* un oggetto, ma fino a che non usciamo, sappiamo di essere sempre in un grande oggetto che è... l'intero programma! E per nostra fortuna il programma ha alcuni suoi propri metodi, come **puts** e **gets**. Date uno sguardo a questo:

```
nonPossoCredereDiAverSceltoUnNomeCosiLungoSoloPerPuntareA3 = 3
puts nonPossoCredereDiAverSceltoUnNomeCosiLungoSoloPerPuntareA3
self.puts nonPossoCredereDiAverSceltoUnNomeCosiLungoSoloPerPuntareA3
```

```
3
3
```

La cosa importante da capire da tutto questo è che ogni metodo viene eseguito da qualche oggetto, anche se non c'è il punto.

Divertenti metodi per le stringhe

Il primo metodo è **`reverse`** , che fornisce l'inverso di una stringa:

```
var1 = 'roma'
var2 = 'enoteca'
var3 = 'Puoi pronunciare questa frase al contrario?'

puts var1.reverse
puts var2.reverse
puts var3.reverse
puts var1
puts var2
puts var3
```

```
amor
acetone
?oirartnoc la esarf atseuq eraicnunorp iouP
roma
enoteca
Puoi pronunciare questa frase al contrario?
```

Come si può vedere, **`reverse`** non inverte la stringa originale; ma ne fa solo una copia inversa. Ecco perché **`var1`** vale ancora **`'roma'`** anche dopo aver chiamato **`reverse`** su **`var1`**.

Un altro metodo sulle stringhe è `length`, che ci dice il numero dei caratteri (spazi inclusi) nella stringa:

```
puts 'Dimmi il tuo nome completo...'
nome = gets.chomp
puts 'Sai che ci sono ' + nome.length + ' caratteri nel tuo    nome,   '   +
nome + '?'
```

```
Dimmi il tuo nome completo...
Simone Ranzato
#<TypeError: can't convert Fixnum into String>
```

Qualcosa è andata storta, e sembra che sia accaduto qualcosa dopo la riga **`name = gets.chomp`**... Avete riconosciuto il problema? Provate a trovarlo.

C'è un problema con **length**: fornisce un numero, ma noi vogliamo una stringa. La soluzione è semplice, dobbiamo solo inserire un **to_s** (e incrociare le dita):

```
puts 'Dimmi il tuo nome completo...'
nome = gets.chomp
puts 'Sai che ci sono ' + nome.length.to_s + ' caratteri nel tuo nome, ' + nome + '?'
```

```
Dimmi il tuo nome completo...
Simone Ranzato
Sai che ci sono 14 caratteri nel tuo nome, Simone Ranzato?
```

Nota: quello è il numero di *caratteri* del mio nome, non il numero di *lettere*.

Ci sono anche alcuni metodi che cambiano il case (maiuscolo e minuscolo) delle stringhe. **upcase** cambia ogni lettera minuscola in maiuscola, e **downcase** cambia le maiuscole in minuscole. **swapcase** scambia il case di ogni lettera della stringa, e infine, **capitalize** è come **downcase**, con la differenza che cambia solo il primo carattere in maiuscolo (se è una lettera).

```
lettere = 'aAbBcCdDeE'
puts lettere.upcase
puts lettere.downcase
puts lettere.swapcase
puts lettere.capitalize
puts ' a'.capitalize
puts lettere
```

```
AABBCCDDEE
aabbccddee
AaBbCcDdEe
Aabbccddee
 a
aAbBcCdDeE
```

Come si può vedere, dalla riga **puts ' a'.capitalize**, il metodo **capitalize** rende maiuscolo il primo *carattere*, non la prima *lettera*. E anche, come abbiamo visto prima, nel chiamare questi metodi, la variabile **lettere** rimane immutata.

Gli ultimi metodi che vedremo servono a formattare le stringhe. Il primo, **center**, aggiunge degli spazi all'inizio e alla fine della stringa al fine di centrarla. Ricapitolando, a **puts** va detto cosa si vuole stampare, a **+** cosa si vuole aggiungere, a **center** in che modo vogliamo che la nostra stringa sia centrata. Quindi se vogliamo centrare le righe di una poesia dobbiamo fare così:

```
lineWidth = 50
puts(                'Old Mother Hubbard'.center(lineWidth))
puts(               'Sat in her cupboard'.center(lineWidth))
puts(         'Eating her curds an whey,'.center(lineWidth))
puts(          'When along came a spider'.center(lineWidth))
```

```
puts(          'Which sat down beside her'.center(lineWidth))
puts('And scared her poor shoe dog away.'.center(lineWidth))
```

```
                  Old Mother Hubbard
                 Sat in her cupboard
              Eating her curds an whey,
               When along came a spider
              Which sat down beside her
          And scared her poor shoe dog away.
```

Gli altri due metodi di formattazione per le stringhe sono **`ljust`** e **`rjust`**, che servono per giustificare a sinistra e a destra. Sono simili a quello per il centro, tranne che riempiono la stringa con degli spazi sulla destra e sulla sinistra rispettivamente. Diamo un'occhiata a tutti e tre in azione:

```
lineWidth = 40
str = '--> text <--'
puts str.ljust  lineWidth
puts str.center lineWidth
puts str.rjust  lineWidth
puts str.ljust (lineWidth/2) + str.rjust (lineWidth/2)
```

```
--> text <--
              --> text <--
                            --> text <--
--> text <--                --> text <--
```

Un paio di cose da provare

- Scrivere un programma `capoArrabbiato`. Si deve bruscamente chiedere ciò che si vuole. Qualunque sia la risposta, il capo arrabbiato dovrebbe gridare di nuovo, e poi licenziare. Per esempio, se si scrive `Voglio un aumento.`, lui urla indietro `COSA SIGNIFICA "VOGLIO UN AUMENTO."?!? SEI LICENZIATO!!`

- E per giocare di più con `center`, `ljust` e `rjust`: scrivere un programma che visualizzerà una Tabella dei Contenuti come questa:

```
                  Table of Contents

Chapter 1:  Numbers                        page 1
Chapter 2:  Letters                       page 72
Chapter 3:  Variables                    page 118
```

Un po' di aritmetica

Gli altri due metodi aritmetici sono **`**`** (elevamento a potenza) e **`%`** (modulo). Per dire "Cinque al quadrato" in Ruby, si può scrivere come `5**2`. È inoltre possibile utilizzare i decimali per la potenza, quindi se si desidera la radice quadrata di cinque si può scrivere `5**0.5`. Il modulo invece restituisce il resto della divisione per un numero. Se per esempio dividiamo 7 per 3, otteniamo 2 con resto 1. Vediamolo in un programma:

```
puts 5**2
puts 5**0.5
puts 7/3
puts 7%3
puts 365%7
```

```
25
2.23606797749979
2
1
1
```

Infine prima di passare a parlare dei numeri casuali c'è un metodo che va menzionato: **`abs`**. Dà semplicemente il valore assoluto di un numero.

```
puts((5-2).abs)
puts((2-5).abs)
```

```
3
3
```

Numeri casuali

Ruby è dotato di un generatore di numeri casuali molto carino. Il metodo per ottenere un numero scelto a caso è **`rand`**. Se si chiama `rand` si ottiene un decimale maggiore o uguale a `0.0` e minore di `1`. Se si dà `rand` ad un intero (`5` per esempio), si otterrà un intero maggiore o uguale a `0` e minore di `5` (quindi cinque possibili numeri, da `0` a `4`).

Vediamo `rand` in azione:

```
puts rand
puts rand
puts rand
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(1))
puts(rand(1))
puts(rand(1))
puts(rand(99999999999999999999999999999999999999))
puts('Il metereologo ha previsto '+rand(101).to_s+'% di possibilita\' di
pioggia,')
puts('ma non ci si puo\' mai fidare di un metereologo.')
```

```
0.422119163813049
0.510323219568254
0.11000613088893
71
71
31
0
0
```

```
0
7402777510280479418369112153292461005 24
Il metereologo ha previsto 4% di possibilita' di pioggia,
ma non ci si puo' mai fidare di un metereologo.
```

Notate che ho usato `rand(101)` per ottenere numeri tra `0` e `100`, e che `rand(1)` restituisce sempre `0`. Non considerare l'intervallo dei valori di ritorno è l'errore più grande che ho visto fare dalla gente con rand; anche programmatori professionisti e anche in prodotti finiti acquistabili. Avevo anche un lettore cd che se impostato su "riproduzione casuale" faceva ascoltare tutte le canzoni ma non l'ultima… (mi chiedo cosa sarebbe successo se avessi messo un cd con un'unica canzone?)

Si potrebbe inoltre volere che `rand` ritornasse gli *stessi* numeri casuali nella stessa sequenza su due segmenti diversi di codice (per esempio, una volta stavo usando un generatore di numeri casuali per creare un mondo generato casualmente per un videogioco, e se ho trovato un mondo che mi piace particolarmente, potrei volerci giocare di nuovo o inviarlo ad un amico). Per fare questo è necessario impostare un *seed*, utilizzando **`srand`**:

```
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

```
24
35
36
58
70

24
35
36
58
70
```

Darà sempre lo stesso risultato ogni volta che si imposta il *seed* con lo stesso numero. Se si vogliono ottenere di nuovo numeri differenti (come quando non abbiamo usato `srand`), basta chiamare `srand 0`. Questo imposta il seed con un numero davvero strano, usando l'ora del computer, fino ai millisecondi.

Oggetto `math`

Diamo infine uno sguardo all'oggetto `math`.

```
puts(Math::PI)
puts(Math::E)
puts(Math.cos(Math::PI/3))
puts(Math.tan(Math::PI/4))
puts(Math.log(Math::E**2))
puts((1 + Math.sqrt(5))/2)
```

```
3.14159265358979
2.71828182845905
0.5
1.0
2.0
1.61803398874989
```

La prima cosa che avrete notato è probabilmente la notazione `::`. Spiegare cosa sia è al di fuori dello scopo di questa guida… è sufficiente sapere che si può usare `Math::PI`, proprio come ci si aspetterebbe di usare.

Come potete vedere, `Math` ha tutte le caratteristiche che ci si aspetta da una calcolatrice scientifica decente. E come sempre, i numeri decimali sono *molto vicini* all'essere la risposta esatta.

6. Controllo del flusso

Anche se questo capitolo è più breve e semplice di quello sui metodi, ci aprirà un intero mondo di possibilità di programmazione. Dopo questo capitolo, saremo in grado di scrivere realmente programmi interattivi; finora abbiamo fatto programmi che *dicono* cose diverse a seconda dell'input da tastiera, ma dopo questo capitolo effettivamente *faranno* anche cose diverse. Ma prima di poter fare ciò, dobbiamo essere in grado di comparare gli oggetti nei nostri programmi.

Metodi di confronto

Per vedere se un oggetto è maggiore o minore di un altro, usiamo i metodi **>** e **<**, come qui:

```
puts 1 > 2
puts 1 < 2
```

```
false
true
```

Nessun problema. Allo stesso modo possiamo trovare se un oggetto è maggiore o uguale (o minore uguale) di un altro con i metodi **>=** e **<=**

```
puts 5 >= 5
puts 5 <= 4
```

```
true
false
```

Infine possiamo vedere se due oggetti sono uguali o no usando **==** (che significa "sono uguali?") e **!=** (che significa "sono diversi?"). È importante non confondere = con ==. = serve a dire ad una variabile di puntare ad un oggetto (assegnazione) e == è per porre la domanda "questi due oggetti sono uguali?".

```
puts 1 == 1
puts 2 != 1
```

```
true
true
```

Ovviamente possiamo anche confrontare le stringhe, che vengono confrontate secondo il loro ordine lessicografico, che praticamente significa il loro ordine nel dizionario. `cane` viene prima di `gatto` nel dizionario, così:

```
puts 'cane' < 'gatto'
```

```
true
```

C'è un problema però: il modo con cui i computer fanno le cose, ovvero ordinano le lettere maiuscole prima di quelle minuscole (questo è il modo con cui memorizzano le lettere nei caratteri, per esempio: prima tutte le lettere maiuscole, poi tutte quelle minuscole). Questo significa che `'Zoo'` verrà prima di `'ape'`, quindi se vogliamo capire quale parola venga veramente prima in un dizionario, assicuriamoci di usare **`downcase`** (o **`upcase`** o **`capitalize`**) in entrambe le parole prima di provare a confrontarle.

Un'ultima nota prima di passare alle istruzioni condizionali: i metodi di confronto non restituiscono le stringhe `'true'` e `'false'`, ma gli oggetti speciali `true` e `false` (ovviamente `true.to_s` ci darà `'true'`, motivo per cui `puts` stampa `'true'`). `true` e `false` saranno usati per tutto il tempo nel…

Istruzioni condizionali

Si tratta di un concetto semplice ma potente:

```
puts 'Ciao, come ti chiami?'
nome = gets.chomp
puts 'Ciao, ' + nome + '.'
if name == 'Lucia'
  puts 'Che bel nome!'
end
```

```
Ciao, come ti chiami?
```

```
Lucia
Ciao, Lucia.
Che bel nome!
```

Ma se mettiamo un nome diverso…

```
Ciao, come ti chiami?
Marta
Ciao, Marta.
```

Se ciò che viene dopo l'`if` è `true`, allora eseguiamo il codice tra l'`if` e l'`end`; se invece è `false` no.

Il codice tra `if` e `true` è rientrato perché così è più semplice tenerne traccia. Potrebbe non sembrare molto d'aiuto in questo semplice programma, ma quando le cose saranno più complesse, farà una grande differenza.

Spesso ci serve un programma che faccia una cosa se l'espressione è `true` e un'altra se è `false`. Questo è ciò che fa l'`else`:

```
puts 'Io sono un indovino. Dimmi il tuo nome:'
nome = gets.chomp
if nome == 'Lucia'
  puts 'Vedo grandi cose nel tuo futuro'
else
  puts 'Il tuo futuro è... Oh no, è tardi!!'
  puts 'Devo proprio andare, scusa!'
end
```

```
Io sono un indovino. Dimmi il tuo nome:
Lucia
Vedo grandi cose nel tuo futuro.
```

Proviamo ora con un nome diverso…

```
Io sono un indovino. Dimmi il tuo nome:
Luigi
Il tuo futuro è... Oh no, è tardi!!
Devo proprio andare, scusa!
```

È un po' come un bivio nel codice: eseguiamo la parte di codice per chi ha `nome == 'Luca'` oppure (`else`) seguiamo l'altra parte?

E come le ramificazioni di un albero, si possono avere ramificazioni su ramificazioni:

```
puts 'Salve, e benvenuto al 7th Grade English.'
puts 'Io sono la Signora Gabbard. Come si chiama?'
nome = gets.chomp

if nome == nome.capitalize
  puts 'Prego si sieda, ' + nome + '.'
else
```

```
    puts nome + '? vuole dire ' + nome.capitalize + ', giusto?'
    puts 'Non sa nemmeno scrivere il suo nome?'
    reply = gets.chomp

    if reply.downcase == 'si'
      puts 'Hmmph! Bene, si sieda!'
    else
      puts 'SE NE VADA!!'
    end
  end
```

```
Salve, e benvenuto al 7th Grade English.
Io sono la Signora Gabbard. Come si chiama?
marco
marco? Vuole dire Marco, giusto?
Non sa nemmeno scrivere il suo nome?
si
Hmmph! Bene si sieda!
```

Bene, oppure…

```
Salve, e benvenuto al 7th Grade English.
Io sono la Signora Gabbard. Come si chiama?
Marco
Prego si sieda, Marco.
```

A volte potrebbe essere difficile cercare di capire dove vanno tutti gli `if`, gli `else` e gli `end`. Così quello che faccio io è scrivere subito l'`end` quando scrivo l'`if`. Così mentre stavo scrivendo il programma sopra, questo è come appariva:

```
puts 'Salve, e benvenuto al 7th Grade English.'
puts 'Io sono la Signora Gabbard. Come si chiama?'
nome = gets.chomp

if nome == nome.capitalize
else
end
```

Quindi l'ho riempito con dei *commenti*, pezzi di codice che il computer ignorerà:

```
puts 'Salve, e benvenuto al 7th Grade English.'
puts 'Io sono la Signora Gabbard. Come si chiama?'
nome = gets.chomp

if nome == nome.capitalize
  # è civile
else
  # si arrabbia
end
```

Qualsiasi cosa dopo **#** è considerata un commento.

Cicli (loop)

Spesso vogliamo che il computer faccia la stessa cosa più e più volte, dopotutto è questo che i computer si suppone facciano bene.

Quando diciamo a un computer di fare qualcosa ripetutamente, dobbiamo anche dire quando fermarsi. I computer non si annoiano, così se non gli diciamo quando fermarsi, non lo fanno e basta. Dobbiamo essere sicuri che ciò non accada dicendo al computer di ripetere certe parti di programma mentre (**`while`**) una certa condizione è vera. Funziona in modo molto simile all'`if`:

```
command = ''

while command != 'bye'
  puts command
  command = gets.chomp
end

puts 'Torna presto!'
```

```

Ciao?
Ciao?
Salve!
Salve!
Piacere di conoscerti.
Piacere di conoscerti.
Oh... che dolce!
Oh... che dolce!
bye
Torna presto!
```

E questo è un ciclo. (Avrete notato la riga bianca all'inizio dell'output; è data dal primo `puts`, prima del primo `gets`.)

I cicli ci permettono di fare tutte cose interessanti, come son sicuro potete immaginare. Tuttavia possono anche causare problemi se si commettono errori. Che cosa succede se il computer viene intrappolato in un loop infinito? In questo caso basta premere `Ctrl + C`.

Un po' di logica

Diamo ancora un'occhiata al nostro primo programma sulle istruzioni condizionali. Cosa succede se mia moglie torna a casa, vede il programma, lo prova, e non le dice che ha un bel nome? Non vorrei ferirle i sentimenti (o dormire sul divano), così riscriviamolo:

```
puts 'Ciao, come ti chiami?'
nome = gets.chomp
puts 'Ciao, ' + nome + '.'
if name == 'Lucia'
  puts 'Che bel nome!'
```

```
else
  if name == 'Maria'
    puts 'Che bel nome!'
  end
end
```

```
Ciao, come ti chiami?
Maria
Ciao, Maria.
Che bel nome!
```

Bene, funziona… ma non è proprio un bel programma. Perché no? Bene, la regola migliore che io abbia mai imparato nella programmazione è *DRY: Don't Repeat Yourself (non ripetere)*, e probabilmente potrei scrivere un libretto sul perché sia una buona regola. Nel nostro caso abbiamo ripetuto la riga `puts 'Che bel nome!'`. Perché questo è un grosso problema? Ebbene, cosa succede se commetto un errore di ortografia quando ho riscritto? Cosa succede se voglio cambiare aggettivo in entrambe le righe? Io sono pigro, ricordate? Fondamentalmente, se voglio che il programma faccia la stessa cosa quando riceve `'Lucia'` o `'Maria'`, allora dovrebbe realmente *fare la stessa cosa*:

```
puts 'Ciao, come ti chiami?'
nome = gets.chomp
puts 'Ciao, ' + nome + '.'
if (name == 'Lucia' or name == 'Maria')
  puts 'Che bel nome!'
end
```

```
Ciao, come ti chiami?
Maria
Ciao, Maria.
Che bel nome!
```

Molto meglio. Per farlo funzionare, ho usato **`or`**. Gli altri operatori logici sono **`and`** e **`not`**. È sempre una buona idea usare le parentesi quando lavoriamo con loro. Vediamo come funzionano:

```
ioSonoChris = true
ioSonoBiondo = false
miPiaceMangiare = true
mangioSassi = false

puts (ioSonoChris and miPiaceMangiare)
puts (miPiaceMangiare and mangioSassi)
puts (ioSonoBiondo and miPiaceMangiare)
puts (ioSonoBiondo and mangioSassi)
puts
puts (ioSonoChris or miPiaceMangiare)
puts (miPiaceMangiare or mangioSassi)
puts (ioSonoBiondo or miPiaceMangiare)
puts (ioSonoBiondo or mangioSassi)
puts
```

```
puts (not ioSonoBiondo)
puts (not ioSonoChris)
```

```
true
false
false
false

true
true
true
false

true
false
```

L'unico di questi che può colpirvi è `or`. Spesso infatti usiamo "o" per dire "uno o l'altro, ma non entrambi". Il computer usa `or` per dire "uno o l'altro, o entrambi", un altro modo di dire sarebbe "almeno uno di questi è vero".

Alcune cose da provare

- Scrivere un programma *Nonna Sorda*. Qualunque cosa dici alla nonna (qualunque cosa digiti), lei risponde con `HUH?! PARLA PIU' FORTE, RAGAZZO MIO!`, fino a che non gli urli (scritto tutto in maiuscolo). Se urli, lei può sentirti (o almeno lo crede) e urla indietro `NO, NON DAL 1938!` Per rendere il programma *davvero* credibile, la nonna deve urlare un anno diverso ogni volta, scelto a caso tra il 1930 e 1950. (Questa parte è opzionale, vedi la parte finale del capitolo sui metodi). Non puoi smettere di parlare alla nonna fino a quando non le urli `CIAO`.
  Suggerimento: *Non dimenticare* `chomp`*!* `'CIAO'` *con un Invio è diverso da* `'CIAO'` *senza!*
  Suggerimento 2: *Provate a pensare a quali parti del vostro programma si ripetono più e più volte. Tutte queste dovrebbero essere nel* ciclo `while`.

- Estendere il programma *Nonna Sorda*: che cosa succede se la nonna non vuole che te ne vai? Quando si dice `CIAO`, può far finta di non sentire. Cambia il programma precedente in modo da doverle urlare `CIAO` tre volte *in riga.* Assicurati di provare il programma: se dici `CIAO` per tre volte, ma non in una riga, dovrai rimanere a parlare con la nonna.

- Anni bisestili. Scrivere un programma dove, richiesto un anno di inizio e uno di fine, `puts` tutti gli anni bisestili nel mezzo (inclusi loro, se sono bisestili). Gli anni bisestili sono anni divisibili per quattro (come il 1984 e il 2004). Tuttavia, gli anni divisibili per 100 *non* sono anni bisestili (ad esempio 1800 e 1900), *a meno* che non siano divisibili per 400 (come il 1600 e il 2000).

7. Array e iteratori

Scriviamo un programma che ci chieda di digitare quante più parole vogliamo (una parola per riga, continuando finché non si digita `Enter` in una riga vuota) e che poi ce le ripeta in ordine alfabetico. Ok?

Bene, non credo ce la possiamo fare. Abbiamo bisogno di un modo di immagazzinare un numero indefinito di parole e come tenerne traccia di tutte insieme, in modo che non si confondano con le altre variabili. Abbiamo bisogno di metterle in una sorta di lista. Abbiamo bisogno degli *array*.

Un array è solo una lista nel computer. Ogni cella nella lista si comporta come una variabile: si può vedere a quale oggetto un particolare slot punti a si può farlo puntare ad un oggetto diverso. Diamo uno sguardo ad alcuni array:

```
[]
[5]
['Salve', 'Arrivederci']

gusto = 'vaniglia'            # Ovviamente questo non è un array
[89.9, gusto, [true, false]] # ...ma questo si.
```

Prima abbiamo un array vuoto, poi un array contenente un solo numero, poi un array contenente due stringhe. Dopo abbiamo una semplice assegnazione, poi un array contenente tre oggetti, l'ultimo dei quali è l'array `[true, false]`. Ricorda, le variabili non sono oggetti, dunque il nostro ultimo array punta realmente ad un decimale, una stringa e un array. Anche se facessimo puntare `gusto` a qualcos'altro l'array non cambierebbe.

Per aiutarci a trovare un particolare oggetto in un array, ogni cella un numero indice. I programmatori iniziano a contare da zero, così la prima cella in un array è la cella zero. Ecco come abbiamo i riferimenti agli oggetti in un array:

```
nomi = ['Ada', 'Bianca', 'Carla']

puts nomi
puts nomi[0]
puts nomi[1]
puts nomi[2]
puts nomi[3]  # Questo è fuori dei limiti.
```

```
Ada
Bianca
Carla
Ada
Bianca
Carla
nil
```

Vediamo che `puts nomi` stampa ciascun nome dell'array `nomi`. Dopo usiamo `puts nomi[0]` per stampare il primo nome nell'array, e `puts nomi[1]` per stampare il secondo, e via dicendo. Sono sicuro che questo vi confonderà, ma ci si abitua. Bisogna solo cominciare a pensare che il conteggio inizia da zero, e non usare parole come "primo" e "secondo". Se ci sono cinque portate, non parlate della prima, ma della portata zero (e nella vostra testa pensate `portata[0]`). Avete cinque dita nella mano destra, e i loro numeri sono 0, 1, 2, 3 e 4.

Infine abbiamo provato `puts nomi[3]`, giusto per vedere cosa sarebbe successo. Vi aspettavate un errore? A volte, quando fate una domanda, la domanda non ha senso (o almeno per il computer), questo è quando si verifica un errore. talvolta, tuttavia, si può porre una domanda e ottenere come risposta *nulla*. Cosa c'è nella cella 3? Nulla. Qual è `nomi[3]`? **`nil`**: il modo di Ruby per dire "nulla". `nil` è un oggetto speciale che praticamente significa "nessun altro oggetto".

Il metodo `each`

`each` ci permette di fare qualcosa (qualsiasi cosa) per ciascun (each) oggetto al quale punta l'array. Quindi, se vogliamo dire qualcosa di carino su ciascuna lingua nell'array sottostante, facciamo così:

```
languages = ['English', 'German', 'Ruby']

languages.each do |lang|
  puts 'I love ' + lang + '!'
  puts 'Tu no?'
end

puts 'E sentiamo sul C++!'
puts '...'
```

```
I love English!
Tu no?
I love German!
Tu no?
I love Ruby!
Tu no?
E sentiamo sul C++!
...
```

Bene, siamo stati in grado di andare in ogni oggetto dell'array senza usare alcun numero, che è decisamente piacevole. Tradotto sarebbe: per ogni (`each`) oggetto in `languages`, punta la variabile `lang` all'oggetto e poi fai (`do`) quello che ti dico, fino a che non arrivi alla fine (`end`).

Potreste pensare "è un po' come i cicli che abbiamo imparato prima". Sì, è simile. Una differenza importante è che il metodo `each` è proprio questo: un metodo. `while` e `end`

(proprio come `do`, `if`, `else` e tutte le altre parole `blu`) non sono metodi. Sono parti fondamentali del linguaggio Ruby, proprio come = e le parentesi.

Metodi come `each` che “si comportano come” i cicli sono spesso chiamati *iteratori*.

Una cosa da sottolineare sugli iteratori è che sono sempre seguiti da `do`…`end`. `while` e `if` non hanno mai un `do` vicino a loro; usiamo do solo con gli iteratori.

Ecco un altro metodo carino, ma non è un metodo sugli array…

```
3.times do
  puts 'Hip-Hip-Hooray!'
end
```

```
Hip-Hip-Hooray!
Hip-Hip-Hooray!
Hip-Hip-Hooray!
```

Altri metodi sugli array

Abbiamo visto `each`, ma ci sono molti altri metodi sugli array… almeno tanti quanti ce ne sono per le stringhe. In effetti, alcuni di loro (come `length`, `reverse`, `+` e `*`) funzionano come per le stringhe, salvo che essi lavorano sulle celle dell’array piuttosto che sulle lettere della stringa. Altri, come **`last`** e **`join`**, sono specifici per gli array. Altri ancora, come **`push`** e **`pop`**, modificano l’array.

In primo luogo diamo un’occhiata a `to_s` e a `join`. `join` funziona in modo simile a `to_s`, tranne per il fatto che aggiunge una stringa tra gli oggetti dell’array. Vediamo:

```
foods = ['carciofo', 'brioche', 'caramello']

puts foods
puts
puts foods.to_s
puts
puts foods.join(', ')
puts
puts foods.join('  :)  ') + '  8)'

200.times do
  puts []
end
```

```
carciofo
brioche
caramello

["carciofo", "brioche", "caramello"]

carciofo, brioche, caramello

carciofo  :)  brioche  :)  caramello  8)
```

Come potete vedere, `puts` tratta gli array in maniera diversa dagli altri oggetti: chiama semplicemente `puts` su ciascun oggetto dell'array; è per questo che `puts` su un array vuoto, ripetuto 200 volte, non fa nulla. Provate a fare `puts` su un array contenente altri array: fa quello che vi aspettavate?

Guardiamo ora `push`, `pop` e `last`. I metodi `push` e pop sono come opposti, come + e -. `push` aggiunge un oggetto alla fine dell'array, e `pop` rimuove l'ultimo oggetto dall'array (mostrando quale è). `last` è simile a `pop` in quanto mostra la fine dell'array, però lasciandolo invariato. `push` e `pop` *modificano* effettivamente l'array:

```
stanze = []
stanze.push 'cucina'
stanze.push 'camera da letto'

puts stanze[0]
puts stanze.last
puts stanze.length

puts stanze.pop
puts stanze
puts stanze.length
```

```
cucina
camera da letto
2
camera da letto
cucina
1
```

Alcune cose da provare

- Scrivere il programma di cui abbiamo parlato all'inizio di questo capitolo. ***Suggerimento:*** *usare il metodo* **`sort`** *che dà la versione ordinata di un array.*

- Provare a scrivere il programma di cui sopra *senza* utilizzare il metodo `sort`. Una gran parte della programmazione è risolvere problemi, così fate più pratica possibile!

- Riscrivere il programma della Tabella di Contenuti (dal capitolo sui metodi). Iniziare il programma con un array che contiene tutte le informazioni della Tabella dei Contenuti (nome del capitolo, numeri di pagina, ecc.). Poi stampare le informazioni dall'array formattando per bene la Tabella dei Contenuti.

8. Scrivere metodi propri

Come abbiamo visto, cicli e iteratori ci permettono di fare la stessa cosa (eseguire lo stesso codice) più e più volte. Tuttavia, a volte vogliamo fare la stessa cosa un certo numero di volte,

ma da parti diverse nel programma. Per esempio, diciamo che stavamo scrivendo un programma con questionario per uno studente di psicologia.

```
puts 'Ciao e grazie per concedermi del tuo tempo'
puts 'per aiutarmi con questo esperimento. Il mio esperimento'
puts 'ha a che fare con cio\' che pensa la gente sul'
puts 'cibo messicano. Basta pensare al cibo messicano'
puts 'e provare a rispondere sinceramente ad ogni domanda,'
puts 'con un "si" o un "no". Il mio esperimento'
puts 'non ha nulla a che fare con la pipi\' a letto.'
puts

# Facciamo queste domande, ma ignoriamo le risposte.

goodAnswer = false
while (not goodAnswer)
  puts 'Ti piacciono i tacos?'
  answer = gets.chomp.downcase
  if (answer == 'si' or answer == 'no')
    goodAnswer = true
  else
    puts 'Per piacere rispondi con "si" or "no".'
  end
end

goodAnswer = false
while (not goodAnswer)
  puts 'Ti piacciono i burritos?'
  answer = gets.chomp.downcase
  if (answer == 'si' or answer == 'no')
    goodAnswer = true
  else
    puts 'Per piacere rispondi con "si" or "no".'
  end
end

# Facciamo attenzione a *questa* risposta.
goodAnswer = false
while (not goodAnswer)
  puts 'Ti fai la pipi\' a letto?'
  answer = gets.chomp.downcase
  if (answer == 'si' or answer == 'no')
    goodAnswer = true
    if answer == 'si'
      wetsBed = true
    else
      wetsBed = false
    end
  else
    puts 'Per piacere rispondi con "si" or "no".'
  end
end
```

```ruby
goodAnswer = false
while (not goodAnswer)
  puts 'Ti piacciono i chimichangas?'
  answer = gets.chomp.downcase
  if (answer == 'si' or answer == 'no')
    goodAnswer = true
  else
    puts 'Per piacere rispondi con "si" or "no".'
  end
end

puts 'Solo un\'altra domanda...'

goodAnswer = false
while (not goodAnswer)
  puts 'Ti piacciono i sopapillas?'
  answer = gets.chomp.downcase
  if (answer == 'si' or answer == 'no')
    goodAnswer = true
  else
    puts 'Per piacere rispondi con "si" or "no".'
  end
end

#  Fare altre domande sul cibo messicano.

puts
puts 'Conclusione:'
puts 'Grazie per avermi concesso del tuo tempo per aiutarmi'
puts 'con questo esperimento. In realta\' questo esperimento'
puts 'non ha nulla a che fare con il cibo messicano.'
puts 'E\' un esperimento sul fare la pipi\' a letto.'
puts 'Il cibo messicano serviva solo per far abbassare la'
puts 'guardia nella speranza di avere risposte piu\' spontanee.'
puts 'Grazie ancora.'
puts
puts wetsBed
```

```
Ciao e grazie per concedermi del tuo tempo
per aiutarmi con questo esperimento. Il mio esperimento
ha a che fare con cio' che pensa la gente sul
cibo messicano. Basta pensare al cibo messicano
e provare a rispondere sinceramente ad ogni domanda,
con un "si" o un "no". Il mio esperimento
non ha nulla a che fare con la pipi' a letto.

Ti piacciono i tacos?
si
Ti piacciono i burritos?
si
Ti fai la pipi' a letto?
assolutamente no
Per piacere rispondi con "si" or "no".
Ti fai la pipi' a letto?
```

```
NO
Ti piacciono i chimichangas?
no
Solo un'altra domanda...
Ti piacciono i sopapillas?
si

Conclusione:
Grazie per avermi concesso del tuo tempo per aiutarmi
con questo esperimento. In realta' questo esperimento
non ha nulla a che fare con il cibo messicano.
E' un esperimento sul fare la pipi' a letto.
Il cibo messicano serviva solo per far abbassare la
guardia nella speranza di avere risposte piu' spontanee.
Grazie ancora.

false
```

Questo era un carino e lungo programma, con molte ripetizioni (tutte le sezioni di codice sulle domande sul cibo messicano erano identiche e quella sulla pipì a letto era differente solo di poco). Le ripetizioni non sono una cosa buona. Non possiamo però metterle in un unico grande ciclo o iteratore, perché a volte ci sono cose che vogliamo vengano fatte tra due domande. In situazioni come queste la soluzione migliore è scrivere un metodo. Ecco come:

```
def scriviMoo
  puts 'mooooooo...'
end
```

```
```

Uh… il nostro programma non ha scritto mooo. Perché no? Perché non gli abbiamo detto di farlo. Gli abbiamo detto *come* farlo, ma non ancora di farlo! Proviamo di nuovo:

```
def scriviMoo
  puts 'mooooooo...'
end

scriviMoo
scriviMoo
puts 'coin-coin'
scriviMoo
scriviMoo
```

```
mooooooo...
mooooooo...
coin-coin
mooooooo...
mooooooo...
```

Molto meglio. Abbiamo definito (**`def`**) il metodo `scriviMoo` (il nome dei metodi, come per le variabili, inizia con la lettera minuscola; eccezioni sono ad esempio + e `==`). Ma i metodi non devono essere sempre associati con gli oggetti? Beh, sì, e in questo caso (come con `puts`

e gets), il metodo è solo associato con l'oggetto rappresentato dall'intero programma. Nel prossimo capitolo vedremo come aggiungere metodi ad altri oggetti. Ma prima…

Parametri dei metodi

Avrete notato che alcuni metodi (come gets, to_s, reverse...) si possono chiamare su un oggetto. Tuttavia, altri metodi (ad esempio +, -, puts...) prendono dei *parametri* per dire all'oggetto come eseguire il metodo. Per esempio, non dovete dire solo 5 +, giusto? State dicendo solo di aggiungere a 5, ma non che *cosa* aggiungere.

Per aggiungere un parametro a scriviMoo (diciamo, il numero di moo), facciamo così:

```
def scriviMoo numberOfMoos
  puts 'mooooooo...'*numberOfMoos
end

scriviMoo 3
puts 'oink-oink'
scriviMoo  # Questo darà un errore perché manca il parametro
```

```
mooooooo...mooooooo...mooooooo...
oink-oink
#<ArgumentError: wrong number of arguments (0 for 1)>
```

numberOfMoos è una variabile che punta al parametro che viene passato. Lo dirò un'altra volta: numberOfMoos è una variabile che punta al parametro che viene passato. Così se scrivo scriviMoo 3, il parametro è 3 e la variabile numberOfMoos punta a 3.

Come potete vedere, il parametro ora è *obbligatorio*. Dopotutto, scriviMoo dovrebbe moltiplicare 'mooooooo...' per che cosa, se non per un parametro? Il vostro povero computer non ne ha idea.

Se gli oggetti in Ruby sono come i nomi e i metodi sono come i verbi, allora potete pensare ai parametri come agli avverbi (come con scriviMoo, dove il parametro ci dice come) o a qualcosa come un complemento oggetto (come con puts).

Variabili locali

Nel programma seguente ci sono due variabili:

```
def doubleThis num
  numTimes2 = num*2
  puts num.to_s+' per due fa '+numTimes2.to_s
end

doubleThis 44
```

```
44 per due fa 88
```

Le variabili sono `num` e `numTimes2`. Entrambe sono dentro al metodo `doubleThis`. Queste (e tutte le variabili che avete visto fin'ora) sono *variabili locali*. Questo significa che vivono all'interno del metodo e non posso uscirne. Se provate, vi verrà dato un errore:

```
def doubleThis num
  numTimes2 = num*2
  puts num.to_s+' per due fa '+numTimes2.to_s
end

doubleThis 44
puts numTimes2.to_s
```

```
44 per due fa 88
#<NameError: undefined local variable or method `numTimes2' for
#<StringIO:0x82ba924>>
```

Variabile locale non definita… In effetti abbiamo definito quella variabile locale, ma non è locale dove abbiamo provato ad usarla, è locale al metodo.

Questo potrebbe sembrare scomodo, ma in realtà è abbastanza interessante. Mentre significa che non avete accesso alle variabili dentro ai metodi, questo significa anche che nessun altro vi ha accesso:

```
def littlePest var
  var = nil
  puts 'HAHA! Ho rovinato la tua variabile!'
end

var = 'Non puoi toccare la mia variabile!'
littlePest var
puts var
```

```
HAHA! Ho rovinato la tua variabile!
Non puoi toccare la mia variabile!
```

Ci sono due variabili chiamate `var` in questo piccolo programma: una all'interno di `littlePest`, e una fuori. Quando abbiamo chiamato `littlePest var`, in realtà gli abbiamo passato la stringa da `var` all'altra, così entrambe puntavano alla stessa stringa. Poi `littlePest` ha puntato la sua `var` *locale* a `nil`, senza far nulla alla `var` fuori del metodo.

Valori di ritorno

Avrete notato che alcuni metodi vi restituiscono qualcosa quando vengono chiamati. Per esempio, `gets` restituisce una stringa (la stringa che avete scritto) e il metodo + in `5 + 3` (che in realtà sarebbe `5.+(3)`) restituisce `8`. I metodi aritmetici per i numeri restituiscono numeri e quelli per le stringhe restituiscono stringhe.

È importante capire la differenza tra i metodi che restituiscono un valore dove il metodo è stato chiamato, e i metodi che stampano informazioni sullo schermo, coma fa `puts`. Notate che `5 + 3` restituisce `8`, non stampa `8`.

Allora che cosa restituisce `puts`? Non ce ne siamo mai interessati, ma guardiamo ora:

```
returnVal = puts 'puts restituisce questo:'
puts returnVal
```

```
puts restituisce questo:
nil
```

Il primo `puts` restituisce `nil`; il secondo, anche se non abbiamo prove, fa lo stesso. Ogni metodo deve restituire qualcosa, anche se si tratta solo di `nil`.

Il valore di ritorno di un metodo è semplicemente l'ultima riga del metodo. Nel caso di `scriviMoo`, significa che ritorna `puts 'mooooooo...'*numberOfMoos`, che è appunto `nil` dato che `puts` ritorna sempre `nil`. Se vogliamo che tutti i nostri metodi restituiscano la stringa `'yellow submarine'` dobbiamo allora mettere questa alla fine:

```
def scriviMoo numberOfMoos
  puts 'mooooooo...'*numberOfMoos
  'yellow submarine'
end

x = scriviMoo 2
puts x
```

```
mooooooo...mooooooo...
yellow submarine
```

`numberOfMoos` è una variabile che punta al parametro che viene passato. Lo dirò un'altra volta:

```
def ask question
  goodAnswer = false
  while (not goodAnswer)
    puts question
    reply = gets.chomp.downcase

    if (reply == 'si' or reply == 'no')
      goodAnswer = true
      if reply == 'si'
        answer = true
      else
        answer = false
      end
    else
      puts 'Please answer "si" or "no".'
    end
  end

  answer  # questo è quello che restituiamo (true or false).
end

puts 'Ciao, e grazie...'
```

```
puts

ask 'Ti piacciono i tacos?'   # ignoriamo il valore di ritorno
ask 'Ti piacciono i burritos?'
wetsBed = ask 'Fai la pipi\' a letto?'  # salviamo cosa ritorna
ask 'Ti piacciono i chimichangas?'
ask 'Ti piacciono i sopapillas?'
ask 'Ti piacciono i tamales?'
puts 'Solo qualche altra domanda...'
ask 'Ti piace bere l\'horchata?'
ask 'Ti piacciono i flautas?'

puts
puts 'Grazie mille...'
puts
puts wetsBed
```

```
Ciao, e grazie...

Ti piacciono i tacos?
si
Ti piacciono i burritos?
no
Fai la pipi' a letto?
no
Ti piacciono i chimichangas?
si
Ti piacciono i sopapillas?
si
Ti piacciono i tamales?
no
Solo qualche altra domanda...
Ti piace bere l'horchata?
si
Ti piacciono i flautas?
no

Grazie mille...

false
```

Un altro grande esempio

Credo potrebbe essere utile un altro esempio sui metodi. Lo chiameremo `englishNumber`. Dato un numero, come 22, lo restituirà in inglese (in questo caso, la stringa `'twenty-two'`). Per ora lavoriamo su numeri interi da `0` a `100`.

Nota: *Questo metodo utilizza la parola chiave* `return` *per uscire prima da un metodo e introduce una nuova opzione di scelta:* `elsif`*. Dovrebbe essere chiaro nel contesto come funzionano.*

```
def englishNumber number
  #  We only want numbers from 0-100.
  if number < 0
    return 'Please enter a number zero or greater.'
  end
  if number > 100
    return 'Please enter a number 100 or lesser.'
  end

  numString = ''  #  This is the string we will return.

  #  "left" is how much of the number we still have left to write out.
  #  "write" is the part we are writing out right now.
  #  write and left... get it?  :)
  left  = number
  write = left/100          #  How many hundreds left to write out?
  left  = left - write*100  #  Subtract off those hundreds.

  if write > 0
    return 'one hundred'
  end

  write = left/10          #  How many tens left to write out?
  left  = left - write*10  #  Subtract off those tens.

  if write > 0
    if write == 1  #  Uh-oh...
      #  Since we can't write "tenty-two" instead of "twelve",
      #  we have to make a special exception for these.
      if    left == 0
        numString = numString + 'ten'
      elsif left == 1
        numString = numString + 'eleven'
      elsif left == 2
        numString = numString + 'twelve'
      elsif left == 3
        numString = numString + 'thirteen'
      elsif left == 4
        numString = numString + 'fourteen'
      elsif left == 5
        numString = numString + 'fifteen'
      elsif left == 6
        numString = numString + 'sixteen'
      elsif left == 7
        numString = numString + 'seventeen'
      elsif left == 8
        numString = numString + 'eighteen'
      elsif left == 9
        numString = numString + 'nineteen'
      end
      #  Since we took care of the digit in the ones place already,
      #  we have nothing left to write.
      left = 0
```

```
    elsif write == 2
      numString = numString + 'twenty'
    elsif write == 3
      numString = numString + 'thirty'
    elsif write == 4
      numString = numString + 'forty'
    elsif write == 5
      numString = numString + 'fifty'
    elsif write == 6
      numString = numString + 'sixty'
    elsif write == 7
      numString = numString + 'seventy'
    elsif write == 8
      numString = numString + 'eighty'
    elsif write == 9
      numString = numString + 'ninety'
    end

    if left > 0
      numString = numString + '-'
    end
  end

  write = left  #  How many ones left to write out?
  left  = 0     #  Subtract off those ones.

  if write > 0
    if    write == 1
      numString = numString + 'one'
    elsif write == 2
      numString = numString + 'two'
    elsif write == 3
      numString = numString + 'three'
    elsif write == 4
      numString = numString + 'four'
    elsif write == 5
      numString = numString + 'five'
    elsif write == 6
      numString = numString + 'six'
    elsif write == 7
      numString = numString + 'seven'
    elsif write == 8
      numString = numString + 'eight'
    elsif write == 9
      numString = numString + 'nine'
    end
  end

  if numString == ''
    #  The only way "numString" could be empty is if
    #  "number" is 0.
    return 'zero'
  end
```

```
  #  If we got this far, then we had a number somewhere
  #  in between 0 and 100, so we need to return "numString".
  numString
end

puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
```

```
zero
nine
ten
eleven
seventeen
thirty-two
eighty-eight
ninety-nine
one hundred
```

Usiamo ora un po’ di array e cerchiamo di fare pulizia di alcune ripetizioni di troppo:

```
def englishNumber number
  if number < 0  #  No negative numbers.
    return 'Please enter a number that isn\'t negative.'
  end
  if number == 0
    return 'zero'
  end

  #  No more special cases!  No more returns!

  numString = ''  #  This is the string we will return.

  onesPlace = ['one',     'two',       'three',    'four',     'five',
               'six',     'seven',     'eight',    'nine']
  tensPlace = ['ten',     'twenty',    'thirty',   'forty',    'fifty',
               'sixty',   'seventy',   'eighty',   'ninety']
  teenagers = ['eleven',  'twelve',    'thirteen', 'fourteen', 'fifteen',
               'sixteen', 'seventeen', 'eighteen', 'nineteen']

  #  "left" is how much of the number we still have left to write out.
  #  "write" is the part we are writing out right now.
  #  write and left... get it?  :)
  left  = number
  write = left/100          #  How many hundreds left to write out?
  left  = left - write*100  #  Subtract off those hundreds.
```

```
  if write > 0
    #  Now here's a really sly trick:
    hundreds  = englishNumber write
    numString = numString + hundreds + ' hundred'
    #  That's called "recursion".  So what did I just do?
    #  I told this method to call itself, but with "write" instead of
    #  "number".  Remember that "write" is (at the moment) the number of
    #  hundreds we have to write out. After we add "hundreds" to "numString",
    #  we add the string ' hundred' after it.  So, for example, if
    #  we originally called englishNumber with 1999 (so "number" = 1999),
    #  then at this point "write" would be 19, and "left" would be 99.
    #  The laziest thing to do at this point is to have englishNumber
    #  write out the 'nineteen' for us, then we write out ' hundred',
    #  and then the rest of englishNumber writes out 'ninety-nine'.

    if left > 0
      #  So we don't write 'two hundredfifty-one'...
      numString = numString + ' '
    end
  end

  write = left/10          #  How many tens left to write out?
  left  = left - write*10  #  Subtract off those tens.

  if write > 0
    if ((write == 1) and (left > 0))
      #  Since we can't write "tenty-two" instead of "twelve",
      #  we have to make a special exception for these.
      numString = numString + teenagers[left-1]
      #  The "-1" is because teenagers[3] is 'fourteen', not 'thirteen'.

      #  Since we took care of the digit in the ones place already,
      #  we have nothing left to write.
      left = 0
    else
      numString = numString + tensPlace[write-1]
      #  The "-1" is because tensPlace[3] is 'forty', not 'thirty'.
    end

    if left > 0
      #  So we don't write 'sixtyfour'...
      numString = numString + '-'
    end
  end

  write = left  #  How many ones left to write out?
  left  = 0     #  Subtract off those ones.

  if write > 0
    numString = numString + onesPlace[write-1]
    #  The "-1" is because onesPlace[3] is 'four', not 'three'.
  end

  #  Now we just return "numString"...
  numString
```

```
end

puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
puts englishNumber(101)
puts englishNumber(234)
puts englishNumber(3211)
puts englishNumber(999999)
puts englishNumber(1000000000000)
```

```
zero
nine
ten
eleven
seventeen
thirty-two
eighty-eight
ninety-nine
one hundred
one hundred one
two hundred thirty-four
thirty-two hundred eleven
ninety-nine hundred ninety-nine hundred ninety-nine
one hundred hundred hundred hundred hundred hundred
```

Molto meglio. Funziona anche con numeri grandi, anche se non proprio così bene come si potrebbe sperare. Per esempio, penso che `'one trillion'` sarebbe un bel valore di ritorno per l'ultimo numero o anche `'one million million'` (anche se tutti e tre sono corretti).

Alcune cose da provare

- Espandere `englishNumber`. Primo, mettete le migliaia. Così dovrebbe tornare `'one thousand'` invece di `'ten hundred'` e `'ten thousand'` invece di `'one hundred hundred'`.

- Espandere ancora `englishNumber`. Ora mettete i milioni, in modo da ottenere `'one million'` invece di `'one thousand thousand'`. Poi provate ad aggiungere i miliardi. Fino a dove si può arrivare?

9. Le classi

Prima abbiamo visto diversi generi, o *classi*, di oggetti: stringhe, interi, decimali, array e qualche oggetto speciale (`true`, `false` e `nil`) che vedremo in seguito. In Ruby, queste classi iniziano sempre per lettera maiuscola: **`String`**, **`Integer`**, **`Float`**, **`Array`**.... In generale se si vuole creare un nuovo oggetto di una classe si usa **`new`**:

```
a = Array.new  + [12345]  # somma di Array
b = String.new + 'hello'  # somma di String
c = Time.new

puts 'a = '+a.to_s
puts 'b = '+b.to_s
puts 'c = '+c.to_s
```

```
a = 12345
b = hello
c = Wed Dec 27 12:24:36 GMT 2006
```

Dato che gli array e le stringhe possono essere creati usando rispettivamente `[...]` e `'...'`, raramente useremo `new` per crearli. (Benché questo non sia evidente dall'esempio, `String.new` crea una stringa vuota, e `Array.new` crea un array vuoto.) Infine, i numeri rappresentano un'eccezione: non è possibile creare un intero con `Integer.new`. Bisogna scrivere l'intero direttamente.

La classe Time

Dunque che roba è la classe `Time`? Gli oggetti `Time` rappresentano momenti di tempo. È possibile aggiungere (o sottrarre) numeri al (o dal) tempo per ottenere nuovi tempi: aggiungendo `1.5` si ha un nuovo tempo con un secondo e mezzo in più:

```
time  = Time.new   #  The moment you got this web page.
time2 = time + 60  #  One minute later.

puts time
puts time2
```

```
Wed Dec 27 12:24:36 GMT 2006
Wed Dec 27 12:25:36 GMT 2006
```

È possibile anche creare un tempo che rappresenta uno specifico momento usando `Time.mktime`:

```
puts Time.mktime(2000, 1, 1)
puts Time.mktime(1976, 8, 3, 10, 11)
```

```
Sat Jan 01 00:00:00 GMT 2000
Tue Aug 03 10:11:00 GMT 1976
```

Si possono confrontare i tempi usando i metodo di confronto (un tempo precedente è minore di un tempo successivo), e se si sottraggono due tempi si ottiene la loro differenza in secondi. Giocateci un po'!

Alcune cose da provare

- Un miliardo di secondi... calcolate l'esatto numero di secondi trascorsi dalla vostra nascita (se potete). Calcolate quando compirete (o forse quando avete compiuto?) un miliardo di secondi. Quindi segnatelo sul vostro calendario.

- Buon compleanno! Chiedete ad una persona in che anno, mese e giorno è nata. Calcolate quanti anni ha e dategli un bel ceffone per ogni compleanno.

La classe Hash

Un'altra utile classe è `Hash`. Un hash è una specie di array: ha un mucchio di posti che possono puntare a diversi oggetti. Negli array i posti sono allineati in una riga e sono tutti numerati (partendo da zero). In un hash i posti non sono allineati (sono ammucchiati tutti insieme), ed è possibile usare *qualsiasi* oggetto per fare riferimento ad essi. È una buona idea usare gli hash quando si vuole tenere traccia di un sacco di cose che non si adattano bene a stare in una lista ordinata. Ad esempio i colori utilizzati per le diverse parti del codice in questo documento:

```
colorArray = []  # come Array.new
colorHash  = {}  # come Hash.new

colorArray[0]          = 'red'
colorArray[1]          = 'green'
colorArray[2]          = 'blue'
colorHash['strings']   = 'red'
colorHash['numbers']   = 'green'
colorHash['keywords'] = 'blue'

colorArray.each do |color|
  puts color
end
colorHash.each do |codeType, color|
  puts codeType + ':  ' + color
end
```

```
red
green
blue
strings:  red
keywords:  blue
numbers:  green
```

Se si usa un array, bisogna ricordarsi che la posizione `0` rappresenta le stringhe, la `1` i numeri, etc. Se si usa un hash invece è tutto molto semplice! Il posto `'strings'` conterrà ovviamente il colore delle stringhe. Niente da ricordare. Avrete sicuramente notato che

quando si usa `each`, gli oggetti dell'hash non vengono fuori nello stesso ordine d'inserimento. (Almeno non è successo quando l'ho scritto. Ma potrebbero esserlo ora... con gli hash non si sa mai). Per tenere le cose in ordine vanno usati gli array, non gli hash.

Sebbene le persone usino delle stringhe per identificare le posizioni negli hash, si può anche usare qualsiasi tipo di oggetti, anche array o altri hash (sebbene non mi venga in mente nessun buon motivo per farlo...):

```
weirdHash = Hash.new

weirdHash[12] = 'monkeys'
weirdHash[[]] = 'emptiness'
weirdHash[Time.new] = 'no time like the present'
```

Hash e array vanno utilizzati per cose differenti; spetta a voi decidere quale scelta è la migliore per il particolare problema.

Estendere le classi

Alla fine dell'ultimo capitolo, avete scritto un metodo per ottenete la parola corrispondente ad un numero intero. Non era un metodo per gli interi; era solo un metodo generico. Non sarebbe bello poter scrivere qualcosa tipo `22.to_eng` invece di `englishNumber 22`? Ecco come farlo:

```
class Integer

  def to_eng
    if self == 5
      english = 'five'
    else
      english = 'fifty-eight'
    end

    english
  end

end

# Proviamolo su una coppia di numeri...
puts 5.to_eng
puts 58.to_eng
```

```
five
fifty-eight
```

Abbiamo definito un metodo per i numeri interi saltando per un momento nella classe `Integer`, definito il metodo lì dentro, e poi saltando fuori di nuovo. Ora tutti gli interi hanno questo (un po' incompleto) metodo. In realtà, se ad esempio non ci piacesse il funzionamento del metodo `to_s` potremmo anche pensare di ridefinirlo in qualche altro modo... ma non è una buona idea! È meglio lasciare i vecchi metodi in pace e farne di nuovi quando si deve fare qualcosa di nuovo.

Finora l'esecuzione di qualsiasi codice o metodo definito è avvenuta nell'oggetto di default "programma". Nel nostro ultimo programma abbiamo abbandonato quell'oggetto per la prima volta e ci siamo messi nella classe `Integer`. Abbiamo definito un metodo (questo lo rende un metodo per numeri interi) e tutti gli interi possono usarlo. All'interno del metodo si usa `self` per fare riferimento all'oggetto (l'intero) che sta usando il metodo.

Creare le classi

Abbiamo visto un gran numero di diverse classe di oggetti. Comunque è semplice arrivare a tipi di oggetti che Ruby non ha. Fortunatamente creare nuove classi è semplice come modificare quelle già esistenti. Diciamo di voler creare dei dadi in Ruby. Ecco come si potrebbe rendere la classe Dado:

```
class Dado

  def lancio
    1 + rand(6)
  end

end

# facciamo una coppia di dadi...
dadi = [Dado.new, Dado.new]

# ...e lanciamoli.
dadi.each do |dado|
  puts dado.lancio
end
```

```
1
3
```

Possiamo definire tutti i tipi di metodi per i nostri oggetti... ma manca qualcosa. Lavorare con questi oggetti ci fa un po' lavorare come con la programmazione prima di aver imparato qualcosa sulle variabili. Guardate i nostri dadi, per esempio. Possiamo lanciarli, e ogni volta ci danno un numero diverso. Ma se decidiamo di tenere traccia di quel numero, dobbiamo creare una variabile che punti a quel numero. Sembra che ogni dado deve *avere* un numero, e che lanciando il dado deve cambiare il numero. Se noi teniamo traccia del dado, non dobbiamo anche tenere traccia del numero che sta mostrando.

Tuttavia, se si tenta di memorizzare il numero che abbiamo lanciato in una variabile (locale) in `lancio`, verrà persa appena `lancio` è finito. Abbiamo bisogno di memorizzare il numero in un tipo diverso di variabile:

Variabili d'istanza

Normalmente quando vogliamo parlare di una stringa, noi lo chiamano semplicemente una *stringa.* Tuttavia, potremmo anche chiamare un *oggetto stringa.* A volte i programmatori potrebbero chiamare *un'istanza della classe* `String`, ma questa è solo un altro modo (e un po' prolisso) di dire *stringa. Un'istanza* di una classe è solo un oggetto di quella classe.

Quindi le variabili di istanza sono variabili solo di un oggetto. Le variabili locali di un metodo durano fino a che il metodo non finisce. Una variabile d'istanza di un oggetto, dall'altra parte, durerà tanto quanto l'oggetto. Per distinguere le variabili di istanza dalle variabili locali, hanno @ davanti al loro nomi:

```
class Dado

  def lancio
    @numberShowing = 1 + rand(6)
  end

  def risultato
    @numberShowing
  end

end

dadi = Dado.new
dadi.lancio
puts dadi.risultato
puts dadi.risultato
dadi.lancio
puts dadi.risultato
puts dadi.risultato
```

```
4
4
6
6
```

Quindi `lancio` lancia i dadi e `risultato` ci dice il numero che viene visualizzato. Tuttavia, proviamo a guardare ciò che mostra prima di aver lanciato i dadi (prima di aver impostato `@numberShowing`)?

```
class Dado

  def lancio
    @numberShowing = 1 + rand(6)
  end

  def risultato
    @numberShowing
  end

end

#  Dal momento che non ho intenzione di utilizzare questo dado
#  di nuovo, non ho bisogno di salvarlo in una variabile,
puts Dado.new.risultato
```

```
nil
```

Hmmm...bene, almeno non ci ha dato un errore. In realtà in questo caso non ha senso per un dado essere "lanciato", o comunque `nil` starebbe a significare questo. Sarebbe bello se

riuscissimo a impostare il nostro nuovo oggetto dadi giusto quando è stato creato. A questo serve **`initialize`**:

```
class Dado

  def initialize
    #  Lancerò solamente il dado, tuttavia volendo
    #  avrebbe potuto fare qualcosa di diverso,
    #  come impostare il dado con 6 risultati.
    lancio
  end

  def lancio
    @numberShowing = 1 + rand(6)
  end

  def risultato
    @numberShowing
  end

end

puts Dado.new.risultato
```

```
4
```

Quando viene creato un oggetto, il suo metodo `initialize` (se è definito) viene sempre chiamato.

Ora l'unica cosa che potrebbe mancare è un modo per impostare quale lato di un dado mostrare... perché non scrivete un metodo `imbroglione` che fa proprio questo! Tornate quando avete finito (e quando avete prova che funziona, ovviamente). Assicuratevi che qualcuno non possa impostare il dado che mostra una settima faccia!

Facciamo un altro esempio. Diciamo che vogliamo fare un semplice animale domestico virtuale, un cucciolo di drago. Come la maggior parte dei cuccioli, dovrebbe essere in grado di mangiare, dormire, e di fare la cacca, il che significa che dovremo essere in grado di dargli da mangiare, metterlo a letto, e portarlo a passeggio. Internamente, il nostro drago avrà bisogno di tenere traccia se è affamato, stanco, o deve andare, ma non saremo in grado di vedere quando interagiamo con il nostro drago, così come non si può chiedere a un bambino umano "Hai fame?". Faremo anche aggiungere qualche modo divertente per poter interagire con il nostro cucciolo di drago, e quando è nato gli daremo un nome (qualunque cosa si passa nel metodo `new` è passata nel metodo `initialize` per voi). Bene, diamo un'occhiata:

```
class Dragon

  def initialize name
    @name = name
    @asleep = false
    @stuffInBelly     = 10  #  He's full.
```

```
    @stuffInIntestine =  0  #  He doesn't need to go.

    puts @name + ' is born.'
  end

  def feed
    puts 'You feed ' + @name + '.'
    @stuffInBelly = 10
    passageOfTime
  end

  def walk
    puts 'You walk ' + @name + '.'
    @stuffInIntestine = 0
    passageOfTime
  end

  def putToBed
    puts 'You put ' + @name + ' to bed.'
    @asleep = true
    3.times do
      if @asleep
        passageOfTime
      end
      if @asleep
        puts @name + ' snores, filling the room with smoke.'
      end
    end
    if @asleep
      @asleep = false
      puts @name + ' wakes up slowly.'
    end
  end

  def toss
    puts 'You toss ' + @name + ' up into the air.'
    puts 'He giggles, which singes your eyebrows.'
    passageOfTime
  end

  def rock
    puts 'You rock ' + @name + ' gently.'
    @asleep = true
    puts 'He briefly dozes off...'
    passageOfTime
    if @asleep
      @asleep = false
      puts '...but wakes when you stop.'
    end
  end

  private
```

```
  #  "private" means that the methods defined here are
  #  methods internal to the object.  (You can feed
  #  your dragon, but you can't ask him if he's hungry.)

  def hungry?
    #  Method names can end with "?".
    #  Usually, we only do this if the method
    #  returns true or false, like this:
    @stuffInBelly <= 2
  end

  def poopy?
    @stuffInIntestine >= 8
  end

  def passageOfTime
    if @stuffInBelly > 0
      #  Move food from belly to intestine.
      @stuffInBelly     = @stuffInBelly     - 1
      @stuffInIntestine = @stuffInIntestine + 1
    else  #  Our dragon is starving!
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + ' is starving!  In desperation, he ate YOU!'
      exit  #  This quits the program.
    end

    if @stuffInIntestine >= 10
      @stuffInIntestine = 0
      puts 'Whoops!  ' + @name + ' had an accident...'
    end

    if hungry?
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + '\'s stomach grumbles...'
    end

    if poopy?
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + ' does the potty dance...'
    end
  end

end
```

```
pet = Dragon.new 'Norbert'
pet.feed
pet.toss
pet.walk
pet.putToBed
pet.rock
pet.putToBed
pet.putToBed
pet.putToBed
pet.putToBed
```

```
Norbert is born.
You feed Norbert.
You toss Norbert up into the air.
He giggles, which singes your eyebrows.
You walk Norbert.
You put Norbert to bed.
Norbert snores, filling the room with smoke.
Norbert snores, filling the room with smoke.
Norbert snores, filling the room with smoke.
Norbert wakes up slowly.
You rock Norbert gently.
He briefly dozes off...
...but wakes when you stop.
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
Norbert does the potty dance...
You put Norbert to bed.
He wakes up suddenly!
Norbert is starving!  In desperation, he ate YOU!
```

Abbiamo visto un paio di cose nuove in questo esempio. Il primo è semplice: **`exit`** termina il programma lì e subito. La seconda è la parola **`private`** che abbiamo messo nel mezzo della definizione della nostra classe. Avrei potuto farne a meno, ma ho voluto rispettare l'idea che certi metodi sono cose che potete fare a un drago, e altri che semplicemente accadono all'interno. Si può pensare a questi come "sotto il cofano": a meno che non siate un meccanico di automobili, tutto ciò che avete realmente bisogno di sapere è il pedale del gas, il pedale del freno, e il volante. Un programmatore potrebbe chiamarli *l'interfaccia pubblica* per la vostra auto. Come il vostro airbag sa quando aprirsi, è interno alla macchina; l'utente medio (autista) non ha bisogno di sapere di questo.

In realtà, per un esempio un po' più concreto in questo senso, avremmo dovuto parlare di come rappresentare un'auto in un videogioco. In primo luogo, si dovete decidere cosa volete far apparire nella vostra interfaccia pubblica, in altre parole, quali metodi la gente dovrebbe

essere in grado di chiamare su uno dei degli oggetti auto? Beh, dovrebbero essere in grado di spingere il pedale del gas e il pedale del freno, ma avrebbero anche bisogno di essere in grado di specificare quanto stanno spingendo il pedale (c'è una grande differenza tra andare fino in fondo e dargli dei colpetti). Poi avrebbero bisogno di essere in grado di guidare, e ancora una volta, di dire quanto stanno sterzando. Suppongo che si poteva fare di più e aggiungere una frizione, i segnali volta, lanciarazzi, postbruciatore, condensatore di flusso, ecc... dipende da che tipo di gioco state facendo.

All'interno di un oggetto auto, però, servirebbero molte altre cose per funzionare, a partire da una velocità, una direzione e una posizione (al massimo di base). Questi attributi verrebbero modificati premendo sui pedali dell'acceleratore o del freno e girando la ruote, naturalmente, ma l'utente non sarebbe in grado di impostare la posizione direttamente. Potrebbe essere utile anche tenere traccia di slittamenti o danni, e così via. Queste sarebbero tutte interne all'oggetto automobile.

Alcune cose da provare

- Effettuare una classe `OrangeTree`. Questa dovrebbe avere un metodo `height` che restituisce la sua altezza, e un metodo `oneYearPasses`, che, quando viene chiamato, da all'albero un anno di età. Ogni anno l'albero cresce in altezza (per quanto pensate che un albero di arancio dovrebbe crescere in un anno), e dopo un certo numero di anni (ancora una volta, su vostra chiamata), l'albero deve morire. Per i primi anni, non dovrebbe produrre frutti, ma dopo un po' sì, e credo che gli alberi più vecchi producano più frutti rispetto ai più giovani… E, naturalmente, si dovrebbe essere in grado di `countTheOranges` (che restituisce il numero di arance sull'albero), e `pickAnOrange` (che riduce la `@orangeCount` di uno e restituisce una stringa che indica come l'arancia è stata deliziosa, oppure dice solo che non ci sono più arance da raccogliere per quest'anno). Assicuratevi che le arance che non vengono raccolte cadano prima dell'anno successivo.
- Scrivere un programma in modo da poter interagire con il drago bambino. Si dovrebbe essere in grado di inserire i comandi come `feed` (mangia) e `walk` (cammina), e tali metodi devono essere chiamati sul drago. Naturalmente, dal momento che quello che si sta inserendo sono solo stringhe, si dovrà avere una sorta di *metodo di spedizione,* dove il tuo programma controlla che stringa è stata inserita e quindi chiama il metodo appropriato.

10. Blocchi e procedure

È sicuramente una delle caratteristiche più interessanti di Ruby. È la capacità di prendere un *blocco* di codice (codice tra `do` e `end`), avvolgerlo in un oggetto (chiamato *proc),* memorizzarlo in una variabile o passarlo a un metodo, ed eseguire il codice nel blocco ogni volta che si vuole (e più di una volta, se si vuole). Quindi è un po' come un metodo di per sé,

salvo che essa non è legata ad un oggetto *(si tratta* di un oggetto), ed è possibile memorizzarlo o passarlo in giro come con qualsiasi oggetto. Credo che sia tempo di un esempio:

```
toast = Proc.new do
  puts 'Cheers!'
end

toast.call
toast.call
toast.call
```

```
Cheers!
Cheers!
Cheers!
```

Così ho creato un proc (che credo sia l'abbreviazione di "procedura", ma di gran lunga più importante, fa rima con "block") che ha tenuto il blocco di codice, poi ho chiamato (`call`) il proc tre volte. Come potete vedere, è un po' come un metodo.

In realtà, è ancora più simile ai metodi che vi ho mostrato, perché i blocchi possono assumere i parametri:

```
doYouLike = Proc.new do |aGoodThing|
  puts 'I *really* like '+aGoodThing+'!'
end

doYouLike.call 'chocolate'
doYouLike.call 'ruby'
```

```
I *really* like chocolate!
I *really* like ruby!
```

Ma allora, perché non usare semplicemente i metodi? Beh, perché ci sono alcune cose che non si possono fare con i metodi. In particolare, non è possibile passare metodi in altri metodi (ma si possono passare proc nei metodi), e i metodi non possono ritornare altri metodi (ma possono tornare proc). Tutto ciò semplicemente perché i proc sono oggetti, i metodi no.

Metodi che ricevono proc

Quando passiamo un proc in un metodo, siamo in grado di controllare come, se, o quante volte lo chiamiamo. Per esempio, diciamo che c'è qualcosa che vogliamo fare prima e dopo un po' di codice che viene eseguito:

```
def doSelfImportantly someProc
  puts 'Everybody just HOLD ON!  I have something to do...'
  someProc.call
  puts 'Ok everyone, I\'m done.  Go on with what you were doing.'
end

sayHello = Proc.new do
  puts 'hello'
end
```

```
sayGoodbye = Proc.new do
  puts 'goodbye'
end

doSelfImportantly sayHello
doSelfImportantly sayGoodbye
```

```
Everybody just HOLD ON!  I have something to do...
hello
Ok everyone, I'm done.  Go on with what you were doing.
Everybody just HOLD ON!  I have something to do...
goodbye
Ok everyone, I'm done.  Go on with what you were doing.
```

Forse non sembra particolarmente favolosa... ma è così. È fin troppo comune nella programmazione avere dei requisiti severi su cosa si deve fare. Se si desidera salvare un file, per esempio, dovete aprire il file, scrivere le informazioni che si desidera avere, e quindi chiudere il file. Se vi dimenticate di chiudere il file, può verificarsi qualche problema. Ma ogni volta che si desidera salvare o caricare un file, bisogna fare la stessa cosa: aprire il file, fare ciò che si vuole *veramente* fare, quindi chiudere il file. È noioso e facile da dimenticare. In Ruby, salvare (o caricare) file funziona in modo simile al codice di cui sopra, in modo che non ci si deve preoccupare di nulla, se non ciò che si desidera salvare (o caricare).

Si possono anche scrivere metodi che determinano quante volte, o anche *se,* chiamare un proc:

```
def maybeDo someProc
  if rand(2) == 0
    someProc.call
  end
end

def twiceDo someProc
  someProc.call
  someProc.call
end

wink = Proc.new do
  puts '<wink>'
end

glance = Proc.new do
  puts '<glance>'
end

maybeDo wink
maybeDo glance
twiceDo wink
twiceDo glance
```

```
<glance>
```

```
<wink>
<wink>
<glance>
<glance>
```

Questi sono alcuni degli usi più comuni di proc che ci permettono di fare cose che semplicemente non avrebbe potuto fare con metodi da solo.

Prima di procedere, diamo un'occhiata a un ultimo esempio. Finora i proc che abbiamo passato sono stati molto simili gli uni agli altri. Questa volta sarà molto diverso, in modo da poter vedere quanto un metodo dipende dai proc passati in esso. Il nostro metodo riceverà qualche oggetto e un proc, e chiamerà il proc su tale oggetto. Se il proc ritorna false, termina; altrimenti chiamiamo il proc con l'oggetto restituito. Continuiamo a farlo fino a che il proc non restituisce false (che era meglio farlo alla fine, o il programma andrà in crash). Il metodo restituisce l'ultimo valore non-false restituito dal proc.

```
def doUntilFalse firstInput, someProc
  input  = firstInput
  output = firstInput

  while output
    input  = output
    output = someProc.call input
  end

  input
end

buildArrayOfSquares = Proc.new do |array|
  lastNumber = array.last
  if lastNumber <= 0
    false
  else
    array.pop                         # Take off the last number...
    array.push lastNumber*lastNumber  # and replace it with its square...
    array.push lastNumber-1           # followed by the next smaller number
  end
end

alwaysFalse = Proc.new do |justIgnoreMe|
  false
end

puts doUntilFalse([5], buildArrayOfSquares).inspect
puts doUntilFalse('I\'m writing this at 3:00 am; someone knock me out!',
alwaysFalse)
```

```
[25, 16, 9, 4, 1, 0]
I'm writing this at 3:00 am; someone knock me out!
```

Il metodo `inspect` è molto simile a `to_s`, salvo che la stringa che restituisce tenta di mostrare il codice Ruby per costruire l'oggetto che gli avete passato. Qui ci mostra l'intero array restituito dalla nostra prima chiamata a `doUntilFalse`.

Metodi che restituiscono proc

Una delle altre cose divertenti che puoi fare con i proc è quello di crearli all'interno dei metodi e restituirli; questo permette ogni sorta di potere nella programmazione. In ogni caso, mi limito solo a parlarne brevemente.

In questo esempio, `compose` prende due proc e restituisce una nuova proc che, quando viene chiamato, chiama il primo proc e passa il suo risultato al secondo.

```
def compose proc1, proc2
  Proc.new do |x|
    proc2.call(proc1.call(x))
  end
end

squareIt = Proc.new do |x|
  x * x
end

doubleIt = Proc.new do |x|
  x + x
end

doubleThenSquare = compose doubleIt, squareIt
squareThenDouble = compose squareIt, doubleIt

puts doubleThenSquare.call(5)
puts squareThenDouble.call(5)
```

```
100
50
```

Si noti che la chiamata a `proc1` deve essere all'interno delle parentesi per `proc2` in modo che venga fatta prima.

Passare blocchi (non proc) all'interno dei metodi

Gran parte del problema è che ci sono tre passi per i quali dobbiamo passare (definire il metodo, fare il proc, e chiamare il metodo con il proc), quando si ha come la sensazione che dovrebbero essere solo due (definire il metodo, e passare il *blocco* di destra nel metodo), dal momento che la maggior parte del tempo non si desidera utilizzare il proc / blocco dopo che si passa nel metodo. Bene, Ruby ha già pensato a tutto! In effetti, lo avete già fatto ogni volta che si utilizzano gli iteratori.

Vediamo un esempio.

```
class Array

  def eachEven(&wasABlock_nowAProc)
    isEven = true    #  We start with "true" because arrays start with 0,
                     #  which is even.
```

```
    self.each do |object|
      if isEven
        wasABlock_nowAProc.call object
      end

      isEven = (not isEven)  #  Toggle from even to odd, or odd to even.
    end
  end

end

['apple', 'bad apple', 'cherry', 'durian'].eachEven do |fruit|
  puts 'Yum!  I just love '+fruit+' pies, don\'t you?'
end

#  Remember, we are getting the even-numbered elements
#  of the array, all of which happen to be odd numbers,
#  just because I like to cause problems like that.
[1, 2, 3, 4, 5].eachEven do |oddBall|
  puts oddBall.to_s+' is NOT an even number!'
end
```

```
Yum!  I just love apple pies, don't you?
Yum!  I just love cherry pies, don't you?
1 is NOT an even number!
3 is NOT an even number!
5 is NOT an even number!
```

Quindi, per passare in un blocco a `eachEven`, tutto quello che abbiamo dovuto fare è stato mettere il blocco dopo il metodo. In questo modo potete passare un blocco in qualsiasi metodo, anche se molti metodi ignoreranno il blocco. Affinché il vostro metodo non ignori il blocco, ma lo prenda e lo trasformi in un proc, inserite il nome del proc alla fine della lista di parametri del vostro metodo, preceduto da una e commerciale (**&**). Questa parte è un po' più difficile, ma non troppo, e dovete fare ciò solo una volta (quando si definisce il metodo). Poi è possibile utilizzare il metodo più e più volte, proprio come i metodi che ricevono blocchi, come `each` e `times` (ricordate `5.times do`...?).

Se siete un po' confusi, ricordate che cosa dovrebbe fare `eachEven`: chiamare il blocco che è stato passato con ogni altro elemento dell'array. Una volta che l'avete scritto e funziona, non c'è bisogno di pensare che cosa sta realmente facendo sotto il cofano; in effetti, questo è esattamente il *motivo per cui* scriviamo metodi come questo: così non dobbiamo poi pensare a come funzionano di nuovo. Li usiamo e basta.

Proviamo a scrivere un metodo che misuri quanto tempo impiegano diverse sezioni di un programma (conosciuto come *profiling* del codice): quindi prenderemo il tempo prima e dopo l'esecuzione di un pezzo di codice, e ne calcoleremo la differenza.

```
def profile descriptionOfBlock, &block
  startTime = Time.now
```

```
  block.call

  duration = Time.now - startTime

  puts descriptionOfBlock+':  '+duration.to_s+' seconds'
end

profile '25000 doublings' do
  number = 1

  25000.times do
    number = number + number
  end

  puts number.to_s.length.to_s+' digits'
end

profile 'count to a million' do
  number = 0

  1000000.times do
    number = number + 1
  end
end
```

```
7526 digits
25000 doublings:  0.186667 seconds
count to a million:  0.506593 seconds
```

Con questo piccolo metodo, possiamo facilmente calcolare la durata di qualsiasi sezione di qualsiasi programma; abbiamo appena messo il codice in un blocco e inviato in `profile`. Cosa potrebbe essere più semplice? Nella maggior parte dei linguaggi, avremmo dovuto aggiungere in modo esplicito il codice di temporizzazione intorno ad ogni sezione che si vuole misurare.

Alcune cose da provare

- *Grandfather Clock*. Scrivere un metodo che prende un blocco e lo chiama una volta per ogni ora della giornata che passa. In questo modo, passando in ingresso il blocco `do puts 'DONG!' end`, rintoccherebbe come un orologio a pendolo. Provare il metodo con un po' di blocchi diversi (compreso quello suggerito).
  Suggerimento: È possibile utilizzare `Time.now.hour` per ottenere l'ora corrente. Tuttavia, questa restituisce un numero compreso tra `0` e `23`, quindi si dovrà modificare per ottenere i normali numeri dell'orologio da `1` a `12`.

- *Program Logger*. Scrivere un metodo chiamato `log`, che prende una stringa di descrizione di un blocco e, naturalmente, un blocco. Simile a `doSelfImportantly`, dovrebbe scrivere (`puts`) una stringa dicendo che ha iniziato il blocco, e un'altra stringa alla fine dicendo che ha finito il blocco, e dicendo anche cosa ha restituito il blocco. Provare il metodo con l'invio di un blocco di codice. All'interno del blocco, mettere

un'altra chiamata a `log`, passando un altro blocco di questo (questo è chiamato *nesting*). In altre parole, l'output dovrebbe essere qualcosa del tipo:

```
Beginning "outer block"...
Beginning "some little block"...
..."some little block" finished, returning:  5
Beginning "yet another block"...
..."yet another block" finished, returning:  I like Thai food!
..."outer block" finished, returning:  false
```

- *Better Logger*. L'uscita di quest'ultimo logger è un po' difficile da leggere, e le cose peggiorerebbero continuando ad usarlo. Sarebbe molto più facile da leggere se si fanno rientrare le righe nei blocchi interni. Per fare ciò, bisogna tenere traccia di quanto profondamente si va con il nesting ogni volta che il logger vuole scrivere qualcosa. Per effettuare questa operazione, si usi una *variabile globale*, una variabile cioè che si può vedere da qualsiasi punto del codice. Per creare una variabile globale, basta precedere il nome della variabile con **$**, come queste: `$global`, `$nestingDepth` e `$bigTopPeeWee`. Alla fine, il logger dovrebbe dare come output qualcosa di questo tipo:

```
Beginning "outer block"...
  Beginning "some little block"...
    Beginning "teeny-tiny block"...
    ..."teeny-tiny block" finished, returning:  lots of love
  ..."some little block" finished, returning:  42
  Beginning "yet another block"...
  ..."yet another block" finished, returning:  I love Indian food!
..."outer block" finished, returning:  true
```

3. Uso di Ruby per creare file html

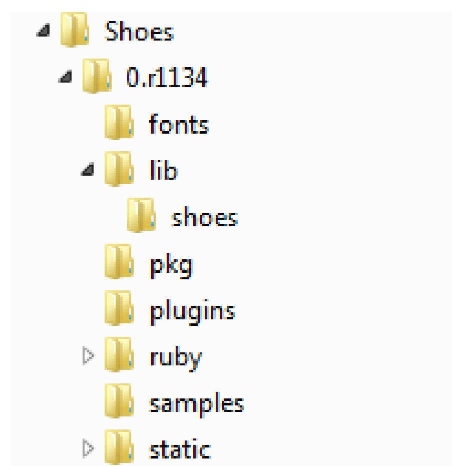
Per la creazione di una documentazione html da un file di testo (.txt) ho analizzato e quindi utilizzato il codice messo a disposizione da Shoes.

Shoes è una piattaforma di toolkit per la scrittura di applicazioni grafiche in modo facile e artistico usando Ruby. A differenza di molti altri toolkit grafici, Shoes è progettato per essere facile e semplice, senza perderne la potenza.

Per scaricare questa piattaforma basta andare sul sito http://shoes.heroku.com/ nella sezione "downloads".

Di fatto Shoes raccoglie poi una serie di applicazioni e di esempi di programmi in Ruby, mettendo anche a disposizione un tutorial per poter creare proprie applicazioni.

Una volta installato Shoes, vediamo che cosa si trova all'interno della cartella `/Shoes/`:



Di particolare interesse sono:

- `shoes.exe`: lancia l'applicazione di Shoes, da dove si possono scegliere gli esempi (contenuti della cartella `/samples/`) e leggere la guida/manuale
- la cartella `/lib/` che contiene il file `shoes.rb` e la cartella `/Shoes/`, tutti file di programma in Ruby.

In particolare il codice necessario per la creazione della documentazione in html è situato in shoes.rb e help.rb. Il testo da cui partire per creare tale documentazione deve essere in formato txt.

Analizziamo ora i segmenti di codice di interesse da shoes.rb:

```
opts.on("--manual-html DIRECTORY", "Saves the manual to a directory as
HTML.") do |dir|
  manual_as :html, dir
  raise SystemExit, "HTML manual in: #{dir}"
end
```

Con il comando `--manual-html DIRECTORY` viene chiamato quindi il metodo `manual_as`, passandogli come parametro `:html` e indicando la cartella di destinazione dei file (`DIRECTORY`). Il metodo, se l'operazione va a buon fine, lancerà il messaggio `"HTML manual in: #{dir}"`, con al posto di `#{dir}` ovviamente il percorso della cartella di destinazione.

Questo invece il metodo che praticamente genera i file html, sempre da shoes.rb:

```
def self.manual_as format, *args
    require 'shoes/search'
    require 'shoes/help'

    case format
    when :shoes
      Shoes.app(:width => 720, :height => 640, &Shoes::Help)
    else
      extend Shoes::Manual
      man = self
      dir, = args
      FileUtils.mkdir_p File.join(dir, 'static')
      FileUtils.cp "static/shoes-icon.png", "#{dir}/static"
      %w[manual.css code_highlighter.js code_highlighter_ruby.js].
        each { |x| FileUtils.cp "static/#{x}", "#{dir}/static" }
      html_bits = proc do
        proc do |sym, text|
        case sym when :intro
          div.intro { p { self << man.manual_p(text, dir) } }
```

```
          when :code
            pre { code.rb text.gsub(/^\s*?\n/, '') }
          when :colors
            color_names = (Shoes::COLORS.keys*"\n").split("\n").sort
            color_names.each do |color|
              c = Shoes::COLORS[color.intern]
              f = c.dark? ? "white" : "black"
              div.color(:style => "background: #{c}; color: #{f}") { h3
color; p c }
            end
          when :index
            tree = man.class_tree
            shown = []
            i = 0
            index_p = proc do |k, subs|
              unless shown.include? k
                i += 1
                p "▸ #{k}", :style => "margin-left: #{20*i}px"
                subs.uniq.sort.each do |s|
                  index_p[s, tree[s]]
                end if subs
                i -= 1
                shown << k
              end
            end
            tree.sort.each &index_p
          #   index_page
          when :list
            ul { text.each { |x| li { self << man.manual_p(x, dir) } } }
          else
            send(TITLES[sym] || :p) { self << man.manual_p(text, dir) }
          end
          end
        end


        #  viene caricato il file, load_docs è un metodo di help.rb
        docs = load_docs(Shoes::Manual::PATH)
        sections = docs.map { |x,| x }

        docn = 1
        docs.each do |title1, opt1|
          subsect = opt1['sections'].map { |x,| x }
          menu = sections.map do |x|
            [x, (subsect if x == title1)]
          end

          path1 = File.join(dir, title1.gsub(/\W/, ''))
          make_html("#{path1}.html", title1, menu) do
            h2 "The Shoes Manual"
            h1 title1
            man.wiki_tokens opt1['description'], true,
&instance_eval(&html_bits)
            p.next { text "Next: "
              a opt1['sections'].first[1]['title'], :href =>
"#{opt1['sections'].first[0]}.html" }
          end
```

```
        optn = 1
        opt1['sections'].each do |title2, opt2|
          path2 = File.join(dir, title2)
          make_html("#{path2}.html", opt2['title'], menu) do
            h2 "The Shoes Manual"
            h1 opt2['title']
            man.wiki_tokens opt2['description'], true,
&instance_eval(&html_bits)
            opt2['methods'].each do |title3, desc3|
              sig, val = title3.split(/\s+»\s+/, 2)
              aname = sig[/^[^(=]+=?/].gsub(/\s/, '').downcase
              a :name => aname
              div.method do
                a sig, :href => "##{aname}"
                text " » #{val}" if val
              end
              div.sample do
                man.wiki_tokens desc3, &instance_eval(&html_bits)
              end
            end
            if opt1['sections'][optn]
              p.next { text "Next: "
                a opt1['sections'][optn][1]['title'], :href =>
"#{opt1['sections'][optn][0]}.html" }
            elsif docs[docn]
              p.next { text "Next: "
                a docs[docn][0], :href => "#{docs[docn][0].gsub(/\W/,
'')}.html" }
            end
            optn += 1
          end
        end

        docn += 1
      end
    end
  end
```

Come si può notare, `manual_as` se riceve come parametro `:shoes` crea la documentazione aprendola come sua applicazione; nel caso invece riceva appunto `:html` la crea in formato html.

Di particolare interesse ritengo siano i tre metodi evidenziati, che troviamo di seguito nel help.rb:

```
module Shoes::Manual
  PATH = "#{DIR}/static/manual.txt"  #  percorso del file .txt
  PARA_RE = /\s*?(\{{3}(?:.+?)\}{3})|\n\n/m
  CODE_RE = /\{{3}(?:\s*\#![^\n]+)?(.+?)\}{3}/m
  IMAGE_RE = /\!(\{([^}\n]+)\})?([^!\n]+\.\w+)\!/
  CODE_STYLE = {:size => 9, :margin => 12}
  INTRO_STYLE = {:size => 16, :margin_bottom => 20, :stroke => "#000"}
  SUB_STYLE = {:stroke => "#CCC", :margin_top => 10}
  IMAGE_STYLE = {:margin => 8, :margin_left => 100}
  COLON = ": "
```

```
  def wiki_tokens(str, intro = false)
    paras = str.split(PARA_RE).reject { |x| x.empty? }
    if intro
      yield :intro, paras.shift
    end
    paras.map do |ps|
      if ps =~ CODE_RE
        yield :code, $1
      else
        case ps
        when /\A\{COLORS\}/
          yield :colors, nil
        when /\A\{INDEX\}/
          yield :index, nil
        when /\A \* (.+)/m
          yield :list, $1.split(/^ \* /)
        when /\A==== (.+) ====/
          yield :caption, $1
        when /\A=== (.+) ===/
          yield :tagline, $1
        when /\A== (.+) ==/
          yield :subtitle, $1
        when /\A= (.+) =/
          yield :title, $1
        else
          yield :para, ps
        end
      end
    end
  end

  def load_docs path
    return @docs if @docs
    str = File.read(path)
    @search = Shoes::Search.new
    @sections, @methods, @mindex = {}, {}, {}
    @docs =
      (str.split(/^= (.+?) =/)[1..-1]/2).map do |k,v|
        sparts = v.split(/^== (.+?) ==/)

        sections = (sparts[1..-1]/2).map do |k2,v2|
          meth = v2.split(/^=== (.+?) ===/)
          k2t = k2[/^(?:The )?([\-\w]+)/, 1]
          meth_plain = meth[0].gsub(IMAGE_RE, '')
          @search.add_document :uri => "T #{k2t}", :body =>
"#{k2}\n#{meth_plain}".downcase

          hsh = {'title' => k2, 'section' => k,
            'description' => meth[0],
            'methods' => (meth[1..-1]/2).map { |_k,_v|
              @search.add_document :uri => "M
#{k}#{COLON}#{k2t}#{COLON}#{_k}", :body => "#{_k}\n#{_v}".downcase
              @mindex["#{k2t}.#{_k[/[\w\.]+/]}"] = [k2t, _k]
              [_k, _v]
          }}
          @methods[k2t] = hsh
          [k2t, hsh]
        end
```

```
        @search.add_document :uri => "S #{k}", :body =>
"#{k}\n#{sparts[0]}".downcase
        hsh = {'description' => sparts[0], 'sections' => sections,
           'class' => "toc" + k.downcase.gsub(/\W+/, '')}
        @sections[k] = hsh
        [k, hsh]
      end
    @search.finish!
    @docs
  end

  def make_html(path, title, menu, &blk)
    require 'hpricot'
    File.open(path, 'w') do |f|
      f << Hpricot do
        xhtml_transitional do
          head do
            meta :"http-equiv" => "Content-Type", "content" =>
"text/html; charset=utf-8"
            title "The Shoes Manual // #{title}"
            script :type => "text/javascript", :src =>
"static/code_highlighter.js"
            script :type => "text/javascript", :src =>
"static/code_highlighter_ruby.js"
            style :type => "text/css" do
              text "@import 'static/manual.css';"
            end
          end
          body do
            div.main! do
              div.manual! &blk
              div.sidebar do
                img :src => "static/shoes-icon.png"
                ul do
                  li { a.prime "HELP", :href => "./" }
                  menu.each do |m, sm|
                    li do
                      a m, :href => "#{m[/^\w+/]}.html"
                      if sm
                        ul.sub do
                          sm.each { |smm| li { a smm, :href =>
"#{smm}.html" } }
                        end
                      end
                    end
                  end

                end
              end
            end
          end
        end
      end.to_html
    end
  end
end
```

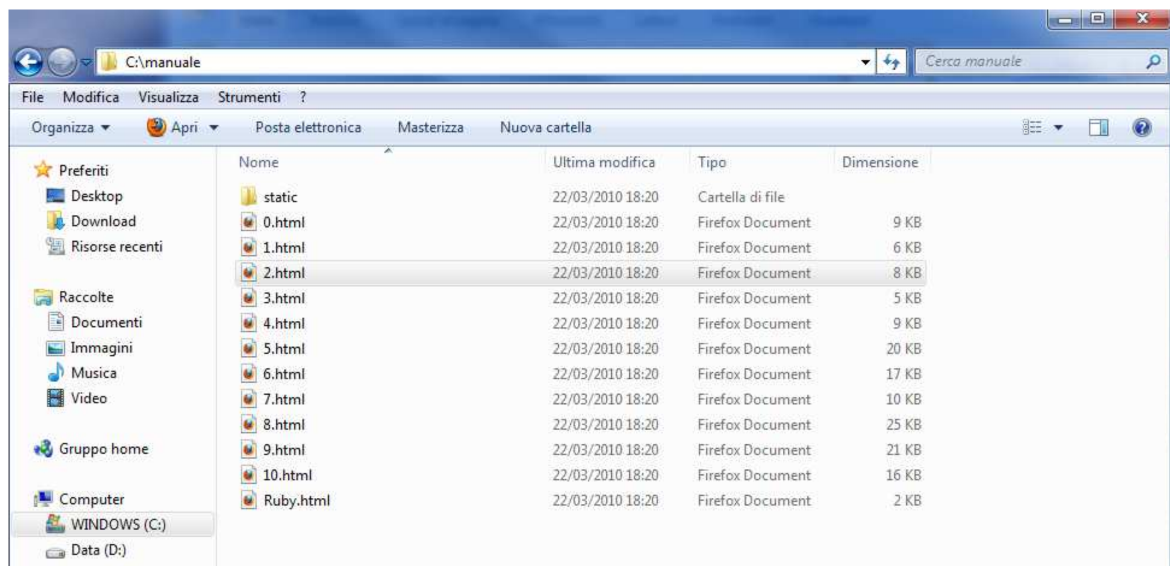
Osservazioni:

- In `PATH` va messo il percorso del file di testo.
- Nel metodo `wiki_tokens` troviamo i caratteri che specificano titoli, sottotitoli, caratteri speciali, ecc.
- Il metodo `load_docs`, come facilmente immaginabile, serve a leggere il documento originale.
- Il metodo `make_html` crea i file in html, utilizzando i file javascript e css specificati nei percorsi.

Ecco dunque che una volta sistemata opportunamente la guida realizzata, con le convenzioni richieste per la realizzazione dei file html, e modificando un po' di percorsi dal codice sopra riportato, lanciando

```
@shoes --manual-html C:\manuale\
```

dal Promp dei Comandi si crea la cartella (nominata `manuale` e situata in `c:\`) con i file della guida in html:



Nella cartella /manuale/static/ si trovano i file manual.css, code_highlighter.js e code_highlighter_ruby.js per gli stili delle pagine (io ho utilizzato quelli di default di Shoes), ed eventuali immagini e file utilizzati nelle varie pagine (come per esempio loghi o file scaricabili).

Diamo ora uno sguardo alle convenzioni del file txt, del quale si vuole realizzare la documentazione.

Come si può vedere nel metodo `wiki_tokens` sopra citato, il titolo è riconosciuto tra due segni di uguale; i titoli dei capitoli sono tra due segni di uguale; ulteriori sottotitoli tra tre e quattro segni di uguale (e con la differenza, rispettivamente, che nel primo caso il sottotitolo avrà uno banda scura dietro, mentre nel secondo cambiano solamente le dimensioni del carattere rispetto a tutto il paragrafo.

```
= titolo =

== capitolo ==

=== sottotitolo1 ===

==== sottotitolo2 ====
```

Altri caratteri utilizzati nella formattazione del testo sono stati:

`'''testo'''` per scrivere in grassetto

`` `testo` `` per indicare le parti di codice

Va inoltre indicato che alcuni caratteri speciali, come le lettere accentate, non vengono riconosciuti.

Per esempio, il seguente segmento di testo:

```
= Ruby =

Realizzazione di una '''guida rapida''' per un primo
approccio a Ruby.

Tale guida ha come testo di riferimento il `Learn to
Program` di Chris Pine, appunto una guida introduttiva
alla programmazione in Ruby.


==== Sommario: ====

0. Come iniziare

1. I numeri

2. Le lettere
```

3. Variabili e assegnazioni

4. Mescoliamoli insieme

5. Qualcos'altro sui metodi

6. Controllo del flusso

7. Array e iteratori

8. Scrivere metodi propri

9. Le classi

10. Blocchi e procedure

== 0. Come iniziare ==

Come iniziare

Quando si programma un computer, e' necessario "parlare" in un linguaggio che il computer capisca: un linguaggio di programmazione. Ci sono decine e decine di linguaggi differenti, e molti di questi sono eccellenti. In questa guida viene trattato il linguaggio di programmazione Ruby.

Quando scriviamo qualcosa nel linguaggio umano, cio' che viene scritto lo chiamiamo testo. Quando scriviamo qualcosa in un linguaggio di programmazione, questo viene invece chiamato codice. Ho inserito molti esempi di codice Ruby nella guida, la maggior parte dei quali sono programmi completi che potete eseguire sul vostro computer.

Ma innanzitutto dobbiamo scaricare e installare Ruby sul nostro computer!

==== Installazione per Windows ====

L'installazione di Ruby per Windows e' velocissima. Come prima cosa bisogna scaricare il programma di installazione di Ruby. Potrebbero esserci piu' di una versione tra le quali scegliere; questa guida usa la versione 1.8.4, quindi assicuriamoci di scaricare una versione recente almeno quanto questa (io suggerirei di scaricare l'ultima versione disponibile, attualmente la 1.9.1). Dopodiche' semplicemente eseguiamo il programma di installazione. Ci verra' chiesto dove installare Ruby e almeno che non si abbia una buona ragione per farlo, si lasci la sua locazione di default.

produce le seguenti pagine:

4. Bibliografia

- Dave Thomas, Programming Ruby – The Pragmaric Programmers' Guide 1.9, Pragmatic Bookshelf, 2009.

- Chris Pine, Learn to Program, Pragmatic Bookshelf, 2006.

- http://www.ruby-lang.org/it/