

Conhecendo Ruby



Eustáquio Rangel

Conhecendo Ruby

Eustáquio Rangel de Oliveira Jr.

Esse livro está à venda em <http://leanpub.com/conhecendo-ruby>

Essa versão foi publicada em 2014-10-23



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2006-2013 Eustáquio Rangel de Oliveira Jr.

Tweet Sobre Esse Livro!

Por favor ajude Eustáquio Rangel de Oliveira Jr. a divulgar esse livro no [Twitter!](#)

O tweet sugerido para esse livro é:

Acabei de comprar/fazer download do ebook "Conhecendo Ruby", do @taq. :-)

A hashtag sugerida para esse livro é [#ruby](#).

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

<https://twitter.com/search?q=#ruby>

Dedicado a minha filha, a luz da minha vida.

Conteúdo

Sobre esse livro	2
Ruby	3
O que é Ruby?	3
Instalando Ruby	3
Instalando via RVM	4
Instalando um interpretador Ruby	7
Básico da linguagem	8
Tipagem dinâmica	8
Tipagem forte	9
Tipos básicos	10
Fixnums	10
Bignums	12
Ponto flutuante	13
Racionais	14
Booleanos	14
Nulos	15
Strings	15
Substrings	16
Concatenando Strings	17
Encoding	18
Váriaveis são referências na memória	19
Congelando objetos	20
Símbolos	21
Expressões regulares	22
Grupos	23
Grupos nomeados	23
Arrays	24
Duck Typing	27
Ranges	27
Hashes	28
Blocos de código	30
Conversões de tipos	31
Conversões de bases	31
Tratamento de exceções	32
Disparando exceções	35

CONTEÚDO

Descobrimo a exceção anterior	35
Criando nossas próprias exceções	35
Utilizando catch e throw	36
Estruturas de controle	38
Condicionais	38
if	38
unless	39
case	40
Loops	41
while	42
for	42
until	43
Operadores lógicos	44
Procs e lambdas	45
Iteradores	48
Selecionando elementos	50
Selecionando os elementos que não atendem uma condição	51
Processando e alterando os elementos	51
Detectando condição em todos os elementos	51
Detectando se algum elemento atende uma condição	52
Detectar e retornar o primeiro elemento que atende uma condição	52
Detectando os valores máximo e mínimo	52
Acumulando os elementos	53
Dividir a coleção em dois Arrays obedecendo uma condição	54
Percorrendo os elementos com os índices	54
Ordenando uma coleção	54
Combinando elementos	55
Percorrendo valores para cima e para baixo	56
Inspecionando no encadeamento de métodos	56
Métodos	57
Retornando valores	57
Enviando valores	58
Enviando e processando blocos e Procs	59
Valores são transmitidos por referência	60
Interceptando exceções direto no método	61
Classes e objetos	64
Classes abertas	70
Aliases	71
Inserindo e removendo métodos	72
Metaclasses	73
Variáveis de classe	76
Interfaces fluentes	78
Variáveis de instância de classe	79
Herança	81

CONTEÚDO

Duplicando de modo raso e profundo	86
Brincando com métodos dinâmicos e hooks	90
Manipulando métodos que se parecem com operadores	92
Closures	97
Módulos	98
Mixins	98
Namespaces	105
TracePoint	108
Instalando pacotes novos através do RubyGems	112
Threads	119
Fibers	129
Continuations	135
Processos em paralelo	136
Benchmarks	141
Entrada e saída	143
Arquivos	143
Arquivos Zip	144
XML	146
XSLT	150
JSON	151
YAML	152
TCP	154
UDP	157
SMTP	158
FTP	159
POP3	160
HTTP	161
HTTPS	162
SSH	163
XML-RPC	164
Python	166
PHP	166
Java	167
JRuby	169
Utilizando classes do Java de dentro do Ruby	170
Usando classes do Ruby dentro do Java	172
Banco de dados	173
Abrindo a conexão	173
Consultas que não retornam dados	173

CONTEÚDO

Atualizando um registro	174
Apagando um registro	175
Consultas que retornam dados	175
Comandos preparados	176
Metadados	177
ActiveRecord	177
Escrevendo extensões para Ruby, em C	180
Garbage collector	186
Isso não é um livro de C mas	187
Isso ainda não é um livro de C, mas	188
Pequeno detalhe: nem toda String usa malloc/free	190
Unit testing	193
Modernizando os testes	196
Randomizando os testes	197
Testando com specs	197
Benchmarks	199
Mocks	200
Stubs	201
Expectations	201
Testes automáticos	204
Criando Gems	206
Criando a gem	206
Testando a gem	208
Construindo a gem	210
Publicando a gem	210
Extraindo uma gem	211
Assinando uma gem	212
Criando um certificado	212
Adaptando a gem para usar o certificado	213
Construindo e publicando a gem assinada	213
Utilizando a gem assinada	213
Gerando documentação	216
Desafios	223
Desafio 1	223
Desafio 2	223
Desafio 3	223
Desafio 4	223
Desafio 5	223
Desafio 6	224
Desafio 7	224

Copyright © 2013 Eustáquio Rangel de Oliveira Jr.

Todos os direitos reservados.

Nenhuma parte desta publicação pode ser reproduzida, armazenada em bancos de dados ou transmitida sob qualquer forma ou meio, seja eletrônico, eletrostático, mecânico, por fotocópia, gravação, mídia magnética ou algum outro modo, sem permissão por escrito do detentor do copyright.

Sobre esse livro

O conteúdo que você tem agora nas mãos é a evolução do meu conhecido “Tutorial de Ruby”, lançado em Janeiro de 2005, que se transformou em 2006 no primeiro livro de Ruby do Brasil, “[Ruby - Conhecendo a Linguagem](#)”¹, da Editora Brasport, cujas cópias se esgotaram (yeah!) e, como não vai ser reimpresso, resolvi atualizar e lançar material nos formatos de *ebook* que agora você tem em mãos.

Quando comecei a divulgar Ruby aqui no Brasil, seja pela internet, seja por palestras em várias cidades, eram poucas pessoas que divulgavam e a linguagem era bem desconhecida, e mesmo hoje, vários anos após ela pegar tração principalmente liderada pela popularidade do *framework Rails*², ainda continua desconhecida de grande parte das pessoas envolvidas ou começando com desenvolvimento de sistemas, especialmente a molecada que está começando a estudar agora em faculdades.

Como eu sou mais teimoso que uma mula, ainda continuo promovendo a linguagem por aí, disponibilizando esse material para estudos, não por causa do tutorial, do livro ou coisa do tipo, mas porque ainda considero a linguagem *muito boa*, ainda mais com toda a evolução que houve em todos esses anos, em que saímos de uma performance mais sofrível (mas, mesmo assim, utilizável) nas versões 1.8.x até os avanços das versões 1.9.x e agora, saindo do forno, a 2.0.

Espero que o material que você tem em mãos sirva para instigar você a conhecer mais sobre a linguagem (aqui não tem nem de longe tudo o que ela disponibiliza) e a conhecer as ferramentas feitas com ela. É uma leitura direta e descontraída, bem direto ao ponto. Em alguns momentos eu forneço alguns “ganchos” para alguma coisa mais avançada do que o escopo atual, e até mostro algumas, mas no geral, espero que seja conteúdo de fácil digestão.

Durante o livro, faço alguns “desafios”, que tem a sua resposta no final do livro. Tentem fazer sem colar! :-)

Um grande abraço!

¹http://books.google.com.br/books?id=rinKvq_oOpkC&redir_esc=y

²<http://rubyonrails.org/>

Ruby

O que é Ruby?

Usando uma pequena descrição encontrada na web, podemos dizer que:

“Ruby é uma linguagem de programação interpretada multiparadigma, de tipagem dinâmica e forte, com gerenciamento de memória automático, originalmente planejada e desenvolvida no Japão em 1995, por Yukihiro “Matz”Matsumoto, para ser usada como linguagem de script. Matz queria uma linguagem de script que fosse mais poderosa do que Perl, e mais orientada a objetos do que Python. Ruby suporta programação funcional, orientada a objetos, imperativa e reflexiva.

Foi inspirada principalmente por Python, Perl, Smalltalk, Eiffel, Ada e Lisp, sendo muito similar em vários aspectos a Python.

A implementação 1.8.7 padrão é escrita em C, como uma linguagem de programação de único passe. Não há qualquer especificação da linguagem, assim a implementação original é considerada de fato uma referência. Atualmente, há várias implementações alternativas da linguagem, incluindo YARV, JRuby, Rubinius, IronRuby, MacRuby e HotRuby, cada qual com uma abordagem diferente, com IronRuby, JRuby e MacRuby fornecendo compilação Just-In-Time e JRuby e MacRuby também fornecendo compilação Ahead-Of-Time.

A série 1.9 usa YARV (Yet Another Ruby VirtualMachine), como também a 2.0 (em desenvolvimento), substituindo a lenta Ruby MRI (Matz’s Ruby Interpreter).”

Fonte: Wikipedia

Instalando Ruby

A instalação pode ser feita de várias maneiras, em diferentes sistemas operacionais, desde pacotes específicos para o sistema operacional, scripts de configuração ou através do download, compilação e instalação do código-fonte. Abaixo vão algumas dicas, mas não execute nenhuma delas pois vamos fazer a instalação de uma maneira diferente.

Se você está usando o Ubuntu, pode instalá-la com os pacotes nativos do sistema operacional:

```
1 $ sudo apt-get install ruby1.9.3
```

Para instalá-la no OSX, pode utilizar o MacPorts:

```
1 $ port install ruby
```

E até no Windows tem um instalador automático. Mais detalhes para esse tipo de instalação podem ser conferidas [no site oficial da linguagem](#)³. Particularmente eu não recomendo utilizar a linguagem no Windows, mas aí vai de cada um.

Instalando via RVM

Vamos instalar Ruby utilizando a [RVM - Ruby Version Manager](#)⁴, que é uma ferramenta de linha de comando que nos permite instalar, gerenciar e trabalhar com múltiplos ambientes Ruby, de interpretadores até conjunto de gems. Como alternativa à RVM, temos também a [rbenv](#)⁵. Vamos utilizar a RVM, mas se mais tarde vocês quiserem investigar a [rbenv](#), fiquem à vontade pois o comportamento é similar.

A instalação da RVM é feita em ambientes que tem o *shell bash* (por isso ela **não está disponível para Windows**, nesse caso, verifique a ferramenta [pik](#)⁶), sendo necessário apenas abrir um terminal rodando esse *shell* e executar:

```
1 $ curl -L https://get.rvm.io | bash
```

Isso irá gerar um diretório em nosso home (abreviado a partir de agora como `~`) parecida com essa:

```
1 $ ls .rvm
2 total 92K
3 drwxr-xr-x 18 taq taq 4,0K 2010-10-14 23:15 .
4 drwxr-xr-x 170 taq taq 12K 2011-06-20 16:38 ..
5 drwxr-xr-x 2 taq taq 4,0K 2010-10-21 15:30 archives
6 drwxr-xr-x 2 taq taq 4,0K 2010-12-14 15:53 bin
7 drwxr-xr-x 2 taq taq 4,0K 2010-12-14 15:53 config
8 drwxr-xr-x 2 taq taq 4,0K 2010-12-14 15:53 environments
9 drwxr-xr-x 2 taq taq 4,0K 2010-10-14 23:05 examples
10 drwxr-xr-x 8 taq taq 4,0K 2010-11-06 13:35 gems
11 drwxr-xr-x 6 taq taq 4,0K 2010-10-14 23:05 gemsets
12 drwxr-xr-x 2 taq taq 4,0K 2010-10-14 23:05 help
13 drwxr-xr-x 3 taq taq 4,0K 2010-10-14 23:05 lib
14 -rw-r--r-- 1 taq taq 1,1K 2010-10-21 15:30 LICENCE
15 drwxr-xr-x 5 taq taq 4,0K 2010-11-06 13:35 log
```

³<http://www.ruby-lang.org/pt/downloads/>

⁴<https://rvm.io/>

⁵<https://github.com/sstephenson/rbenv>

⁶<https://github.com/vertiginous/pik/>

```
16 drwxr-xr-x 5 taq taq 4,0K 2010-10-14 23:05 patches
17 -rw-r--r-- 1 taq taq 6,6K 2010-10-21 15:30 README
18 drwxr-xr-x 4 taq taq 4,0K 2010-12-14 15:53 rubies
19 drwxr-xr-x 3 taq taq 4,0K 2010-10-21 15:30 scripts
20 drwxr-xr-x 7 taq taq 4,0K 2010-10-21 15:30 src
21 drwxr-xr-x 2 taq taq 4,0K 2010-10-14 23:05 tmp
22 drwxr-xr-x 6 taq taq 4,0K 2010-12-14 15:53 wrappers
```

e também com o diretório de gems:

```
1 $ ls .gem
2 total 28K
3 drwxr-xr-x 4 taq taq 4,0K 2010-05-18 16:55 .
4 drwxr-xr-x 170 taq taq 12K 2011-06-20 16:38 ..
5 -rw-r--r-- 1 taq taq 56 2010-05-18 16:55 credentials
6 drwxr-xr-x 3 taq taq 4,0K 2009-08-08 19:26 ruby
7 drwxr-xr-x 6 taq taq 4,0K 2010-11-16 00:14 specs
```

Após a instalação, **dependendo da versão da RVM que foi instalada**, temos que inserir o comando `rvm` no *path*, adicionando no final do arquivo `/.bash_profile`:

```
1 echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm"
2 # Load RVM function' >> ~/.bash_profile
```

Na versão corrente, isso não é mais necessário. Para confirmar se é necessário ou se a RVM já se encontra corretamente configurada e instalada, podemos executar os seguintes comandos:

```
1 $ type rvm | head -n1
2 rvm é uma função
3 $ rvm -v
4 rvm 1.0.15 by Wayne E. Seguin (wayneesequin@gmail.com) [http://rvm.beginrescu\
5 eend.com/]
```

E **dependendo da versão da RVM instalada**, devemos verificar quais são as notas para o ambiente que estamos instalando a RVM, que no caso do Ubuntu vai retornar:

```

1 $ rvm notes
2 Notes for Linux ( DISTRIB_ID=Ubuntu
3 DISTRIB_RELEASE=11.04
4 DISTRIB_CODENAME=natty
5 DISTRIB_DESCRIPTION="Ubuntu 11.04" )
6 # NOTE: MRI stands for Matz's Ruby Interpreter (1.8.X, 1.9.X),
7 # ree stands for Ruby Enterprise Edition and rbx stands for Rubinius.
8 # curl is required.
9 # git is required.
10 # patch is required (for ree, some ruby head's).
11 # If you wish to install rbx and/or any MRI head (eg. 1.9.2-head)
12 # then you must install and use rvm 1.8.7 first.
13 # If you wish to have the 'pretty colors' again,
14 # set 'export rvm_pretty_print_flag=1' in ~/.rvmrc.
15 dependencies:
16 # For RVM
17 rvm: bash curl git
18 # For JRuby (if you wish to use it) you will need:
19 jruby: aptitude install curl sun-java6-bin sun-java6-jre
20 sun-java6-jdk
21 # For MRI & ree (if you wish to use it) you will need
22 # (depending on what you # are installing):
23 ruby: aptitude install build-essential bison openssl libreadline5
24 libreadline-dev curl git zlib1g zlib1g-dev libssl-dev libsqlite3-0
25 libsqlite3-dev sqlite3 libxml2-dev
26 ruby-head: git subversion autoconf
27 # For IronRuby (if you wish to use it) you will need:
28 ironruby: aptitude install curl mono-2.0-devel

```

No caso do Ubuntu e da versão retornar esse tipo de informação, devemos executar a seguinte linha recomendada, em um terminal:

```

1 sudo aptitude install build-essential bison openssl libreadline5
2 libreadline-dev curl git zlib1g zlib1g-dev libssl-dev libsqlite3-0
3 libsqlite3-dev sqlite3 libxml2-dev

```

Desse modo, satisfazemos todas as ferramentas necessárias para utilizar a RVM. Apesar dela citar nas instruções o aptitude, podemos usar sem problemas o apt-get.

Nas últimas versões da RVM, executando

```
1 rvm requirements
```

vão ser instaladas as dependências necessárias, talvez requisitando acesso à mais permissões utilizando o sudo.

Instalando um interpretador Ruby

Após instalar a RVM e suas dependências, agora é hora de instalarmos um interpretador Ruby. Vamos utilizar a versão 1.9.3. Podemos rodar:

```
1 $ rvm install 1.9.3
2 Installing Ruby from source to:
3 /home/taq/.rvm/rubies/ruby-1.9.3-p385,
4 this may take a while depending on your cpu(s)...
5 ruby-1.9.3-p385 - #fetching
6 ruby-1.9.3-p385 - #downloading ruby-1.9.3-p385, this may
7 take a while depending on your connection...
8 ...
```

Após instalado, temos que ativar a versão na RVM e verificar se ficou ok:

```
1 $ rvm 1.9.3
2 $ ruby -v
3 ruby 1.9.3p385 (2013-02-06 revision 39114) [i686-linux]
```

Uma coisa que enche o saco é ficar toda santa hora indicando qual a versão que queremos rodar. Para evitar isso, vamos deixar a 1.9.3 como versão padrão do sistema:

```
1 $ rvm use 1.9.3 --default
2 Using /home/taq/.rvm/gems/ruby-1.9.3-p385
```

Com tudo instalado e configurado, podemos prosseguir.

Básico da linguagem

Vamos conhecer agora alguns dos recursos, características, tipos e estruturas básicas da linguagem. Eu sempre cito em palestras e treinamentos uma frase do [Alan Perlis](#)⁷, que é:

A language that doesn't affect the way you think about programming, is not worth knowing.

Ou, traduzindo:

Não compensa aprender uma linguagem que não afeta o jeito que você pensa sobre programação.

O que vamos ver (pelo menos é a minha intenção) é o que Ruby tem de diferente para valer a pena ser estudada. Não vamos ver só como os `if's` e `while's` são diferentes, mas sim meios de fazer determinadas coisas em que você vai se perguntar, no final, “por que a minha linguagem preferida X não faz isso dessa maneira?”.

Tipagem dinâmica

Ruby é uma linguagem de tipagem dinâmica. Como mencionado na Wikipedia:

Tipagem dinâmica é uma característica de determinadas linguagens de programação, que não exigem declarações de tipos de dados, pois são capazes de escolher que tipo utilizar dinamicamente para cada variável, podendo alterá-lo durante a compilação ou a execução do programa.

Algumas das linguagens mais conhecidas a utilizarem tipagem dinâmica são: Python, Ruby, PHP e Lisp. A tipagem dinâmica contrasta com a tipagem estática, que exige a declaração de quais dados poderão ser associados a cada variável antes de sua utilização. Na prática, isso significa que:

```
1 v = "teste"
2 v.class => String
3 v = 1
4 v.class => Fixnum
```

Pudemos ver que a variável ⁸ `v` pode assumir como valor tanto uma `String` como um número (que nesse caso, é um `Fixnum` - mais sobre classes mais adiante), ao passo, que, em uma linguagem de tipagem estática, como Java, isso não seria possível, com o **compilador** já não nos deixando prosseguir:

⁷http://pt.wikipedia.org/wiki/Alan_Perlis

⁸Variáveis são referências para áreas da memória

```
1 public class Estatica {
2     public static void main(String args[]) {
3         String v = "teste";
4         System.out.println(v);
5         v = 1;
6     }
7 }
8
9 $ javac Estatica.java
10 Estatica.java:5: incompatible types
11 found   : int
12 required: java.lang.String
13 v = 1;
14 ^
15 1 error
```



Existe alguma discussão que linguagens com tipagem estática oferecem mais “segurança”, para o programador, pois, como no caso acima, o compilador executa uma crítica no código e nos aponta se existe algum erro. Particularmente, eu acredito que hoje em dia é delegada muita coisa desse tipo para os compiladores e ferramentas do tipo, removendo um pouco de preocupação do programador, sendo que também não adianta checar tipos se no final o comportamento do seu código pode não corresponder com o que é necessário dele, que é o cerne da questão. Nesse caso, prefiro utilizar metodologias como *Test Driven Development*, que vamos dar uma breve olhada mais para o final do livro.

Também existe a questão de performance, que o código compilado é muito mais rápido que o interpretado. Ruby melhorou **muito** na questão de performance nas versões 1.9 e 2.0, não a ponto de bater código compilado e *linkado* para a plataforma em que roda, mas hoje em dia a não ser que o seu aplicativo exija **muita** performance, isso não é mais um problema. Inclusive, podemos até rodar código Ruby na VM do Java, como veremos mais adiante com o uso de JRuby, e utilizar algumas técnicas que vão deixar Ruby dezenas de vezes mais rápida que Java, onde a implementação da mesma técnica seria dezenas de vezes mais complicada.

Tipagem forte

Ruby também tem tipagem forte. Segundo a Wikipedia:

Linguagens implementadas com tipos de dados fortes, tais como Java e Pascal, exigem que o tipo de dado de um valor seja do mesmo tipo da variável ao qual este valor será atribuído.

Isso significa que:

```
1 i = 1
2 s = "oi"
3 puts i+s
4 TypeError: String can't be coerced into Fixnum
```

Enquanto em uma linguagem como PHP, temos tipagem fraca:

```
1 <?php
2     $i=1;
3     $s="oi";
4     print $i+$s;
5 ?>
```

Rodando isso, resulta em:

```
1 $ php tipagem_fraca.php
2 1
```

Tipos básicos

Não temos primitivos em Ruby, somente abstratos, onde todos exibem comportamento de objetos. Temos números inteiros e de ponto flutuante, onde podemos dividir os inteiros em `Fixnums` e `Bignums`, que são diferentes somente pelo tamanho do número, sendo convertidos automaticamente. Vamos ver alguns deles agora.

Fixnums

Os `Fixnums` são números inteiros de 31 bits de comprimento (ou 1 *word* do processador menos 1 bit), usando 1 bit para armazenar o sinal e 1 bit para indicar que a referência corrente é um `Fixnum` (mais sobre isso logo abaixo, mencionando *immediate values*), resultando em um valor máximo de armazenamento, para máquinas de 32 bits, de 30 bits, ou seja:

```
1 (2**30)-1 => 1073741823
2 ((2**30)-1).class => Fixnum
```

Vamos testar isso no IRB, o interpretador de comandos do Ruby. Para acionar o IRB, abra um emulador de terminal e digite:

```
1 $ irb
2 ruby-1.9.3-p385 > (2**30)-1
3 => 1073741823
4 ruby-1.9.3-p385 > ((2**30)-1).class
5 => Fixnum
```

**Dica**

Podemos descobrir o tipo de objeto que uma variável aponta utilizando o método `class`, como no exemplo acima.

Os `Fixnums` tem características interessantes que ajudam na sua manipulação mais rápida pela linguagem, que os definem como `immediate values`, que são tipos de dados apontados por variáveis que armazenam seus valores na própria referência e não em um objeto que teve memória alocada para ser utilizado, agilizando bastante o seu uso. Para verificar isso vamos utilizar o método `object_id`.

**Dica**

Todo objeto em Ruby pode ser identificado pelo seu `object_id`.

Por exemplo:

```
1 ruby-1.9.3-p385 > n = 1234
2 => 1234
3 ruby-1.9.3-p385 > n.object_id
4 => 2469
5 ruby-1.9.3-p385 > n.object_id >> 1
6 => 1234
```

Também podemos notar que esse comportamento é sólido verificando que o `object_id` de várias variáveis apontando para um mesmo valor continua sendo o mesmo:

```
1 n1 = 1234
2 => 1234
3 n2 = 1234
4 => 1234
5 n1.object_id
6 => 2469
7 n2.object_id
8 => 2469
```

Os `Fixnums`, como `immediate values`, também tem também uma característica que permite identificá-los entre os outros objetos rapidamente através de uma operação de `and` lógico, onde é verificado justamente o último bit, que é um dos 2 utilizados para controle, como explicado anteriormente:

```

1 n = 1234
2 => 1234
3 n.object_id & 0x1
4 => 1

```

Isso nos mostra um comportamento interessante: qualquer variável que aponta para um objeto ou algo como um Fixnum ou immediate value, que apesar de carregar o seu próprio valor e ser bem *light weight*, ainda mantém características onde podem ter acessados os seus métodos como qualquer outro tipo na linguagem. Olhem, por exemplo, o número 1 (e qualquer outro número):

```

1 1.methods => [:to_s, :-, :, :-, :, :, :div, :, :modulo, :divmod, :fdiv, :,
2 :abs, :magnitude, :, :, :=>, :, :, :, :, :, :, :[], :, :, :to_f,
3 :size, :zero?, :odd?, :even?, :succ, :integer?, :upto, :downto, :times,
4 :next, :pred, :chr, :ord, :to_i, :to_int, :floor, :ceil, :truncate,
5 :round, :gcd, :lcm, :gcdlcm, :numerator, :denominator, :to_r,
6 :rationalize, :singleton_method_added, :coerce, :i, :, :eql?, :quo,
7 :remainder, :real?, :nonzero?, :step, :to_c, :real, :imaginary,
8 :imag, :abs2, :arg, :angle, :phase, :rectangular, :rect,
9 :polar, :conjugate, :conj, :pretty_print_cycle, :pretty_print,
10 :between?, :po, :poc, :pretty_print_instance_variables,
11 :pretty_print_inspect, :nil?, :, :!, :hash, :class, :singleton_class,
12 :clone, :dup, :initialize_dup, :initialize_clone, :taint, :tainted?,
13 :untaint, :untrust, :untrusted?, :trust, :freeze, :frozen?, :inspect,
14 :methods, :singleton_methods, :protected_methods, :private_methods,
15 :public_methods, :instance_variables, :instance_variable_get,
16 :instance_variable_set, :instance_variable_defined?, :instance_of?,
17 :kind_of?, :is_a?, :tap, :send, :public_send, :respond_to?,
18 :respond_to_missing?, :extend, :display, :method, :public_method,
19 :define_singleton_method, :__id__, :object_id, :to_enum, :enum_for,
20 :pretty_inspect, :ri, :equal?, :!, :!, :instance_eval,
21 :instance_exec, :__send__]

```

No exemplo acima, estamos vendo os métodos públicos de acesso de um Fixnum. Mais sobre métodos mais tarde!

Bignums

Como vimos acima, os Fixnums tem limites nos seus valores, dependendo da plataforma. Os Bignums são os números inteiros que excedem o limite imposto pelos Fixnums, ou seja:

```
1 (2**30)
2 => 1073741824
3 (2**30).class
4 => Bignum
```

Uma coisa muito importante nesse caso, é que os Bignums **alocam memória**, diferentemente dos Fixnums e outros tipos que são `immediate values`!

Podemos ver isso criando algumas variáveis apontando para o mesmo valor de Bignum e vendo que cada uma tem um `object_id` diferente:

```
1 b1 = (2**30)
2 => 1073741824
3 b2 = (2**30)
4 => 1073741824
5 b1.object_id
6 => 75098610
7 b2.object_id
8 => 75095080
```

Tanto para Fixnums como para Bignums, para efeito de legibilidade, podemos escrever os números utilizando o sublinhado (`_`) como separador dos números:

```
1 1_234_567
2 => 1234567
3 1_234_567_890
4 => 1234567890
```

Ponto flutuante

Os números de ponto flutuante podem ser criados utilizando ... ponto, `dã`. Por exemplo:

```
1 1.23
2 => 1.23
3 1.234
4 => 1.234
5 Listagem 1.9: Ponto flutuante
```

Importante notar que os Floats **não são** `immediate values`:

```
1 f1 = 1.23
2 => 1.23
3 f2 = 1.23
4 => 1.23
5 f1.object_id
6 => 84116780
7 f2.object_id
8 => 84114210
```

Racionais

Podemos criar racionais utilizando explicitamente a classe `Rational`:

```
1 $ Rational(1,3)
2 => 13
3 Rational(1,3).to_s
4 => "1/3"
5 Rational(1,3) * 9
6 => 3
```

Ou, a partir da versão 1.9 de Ruby, utilizar `to_r` em uma `String`:

```
1 $ "1/3".to_r * 9
2 => 3
```



Dica

Podemos ver alguns comportamentos estranhos como:

```
1 "1/3".to_r * 10
2 => 103
```

Nesses casos, é melhor sempre utilizar um número de ponto flutuante:

```
1 "1/3".to_r * 10.0
2 => 3.3333333333333333
```

Booleanos

Temos mais dois `immediate values` que são os *booleanos*, os tradicionais `true` e `false`, indicando como `object_ids`, respectivamente, 2 e 0:

```
1 true.object_id
2 => 2
3 false.object_id
4 => 0
```

Nulos

O tipo nulo em Ruby é definido como `nil`. Ele também é um `immediate value`, com o valor de 4 no seu `object_id`:

```
1 nil.object_id
2 => 4
```

Temos um método para verificar se uma variável armazena um valor nulo, chamado `nil?`:

```
1 v = 1
2 => 1
3 v.nil?
4 => false
5 v = nil
6 => nil
7 v.nil?
8 => true
```

Strings

Strings são cadeias de caracteres, que podemos criar delimitando esses caracteres com aspas simples ou duplas, como por exemplo “azul” ou ‘azul’, podendo utilizar simples ou duplas dentro da outra como “o céu é ‘azul’” ou ‘o céu é “azul”’ e “*escapar*” utilizando o caracter `\`:

```
1 "o céu é 'azul'"
2 => "o céu é 'azul'"
3 'o céu é "azul"'
4 => "o céu é \"azul\""
5 "o céu é \"azul\""
6 => "o céu é \'azul\'"
7 'o céu é \'azul\''
```

Também podemos criar Strings longas, com várias linhas, usando o conceito de heredoc:

```
1 str = <<<FIM
2 criando uma String longa
3 com saltos de linha e
4 vai terminar logo abaixo.
5 FIM
6 => "criando uma String longa\ncom saltos de linha e \nvai terminar logo abaix\
7 o.\n"
```

Para cada String criada, vamos ter espaço alocado na memória, tendo um `object_id` distinto para cada uma:

```
1 s1 = "ola"
2 => "ola"
3 s2 = "ola"
4 => "ola"
5 s1.object_id
6 => 84291220
7 s2.object_id
8 => 84288510
```

Substrings

São pedaços de uma String. Para pegar algumas *substrings*, podemos tratar a String como um Array:

```
1 str = "string"
2 => "string"
3 str[0..2]
4 => "str"
5 str[3..4]
6 => "in"
7 str[4..5]
8 => "ng"
```

Podendo também usar índices negativos:

```
1 str[-4..3]
2 => "ri"
3 str[-5..2]
4 => "tr"
5 str[-4..-3]
6 => "ri"
7 str[-3..-1]
8 => "ing"
9 str[-1]
10 => "g"
11 str[-2]
12 => "n"
```

Ou utilizar o método `slice`, com um comportamento um pouco diferente:

```
1 str.slice(0,2)
2 => "st"
3 str.slice(3,2)
4 => "in"
```

Referenciando um caracter da `String`, temos algumas diferenças entre as versões 1.8.x e 1.9.x (ou maiores) do Ruby:

```
1 # Ruby 1.8.x
2 str[0]
3 => 115
4
5 # Ruby 1.9.x
6 str[0]
7 => "s"
```

Mais sobre *encodings* logo abaixo.

Concatenando Strings

Para concatenar Strings, podemos utilizar os métodos (sim, métodos, vocês não imaginam as bruxarias que dá para fazer com métodos em Ruby, como veremos adiante!) `+` ou `<<`:

```
1 nome = "Eustaquio"
2 => "Eustaquio"
3 sobrenome = "Rangel"
4 => "Rangel"
5 nome + " " + sobrenome
6 => "Eustaquio Rangel"
7 nome.object_id
8 => 84406670
9 nome << " "
10 => "Eustaquio "
11 nome << sobrenome
12 => "Eustaquio Rangel"
13 nome.object_id
14 => 84406670
```

A diferença é que `+` nos retorna um novo objeto, enquanto `<<` faz uma realocação de memória e trabalha no objeto onde o conteúdo está sendo adicionado, como demonstrado acima.

Encoding

A partir da versão 1.9 temos suporte para *encodings* diferentes para as Strings em Ruby. Nas versões menores, era retornado o valor do caracter na tabela ASCII. Utilizando um *encoding* como o UTF-8, podemos utilizar (se desejado, claro!) qualquer caracter para definir até nomes de métodos!



Dica

Para utilizarmos explicitamente um *encoding* em um arquivo de código-fonte Ruby, temos que especificar o encoding logo na primeira linha do arquivo, utilizando, por exemplo, com UTF-8:

```
1 # encoding: utf-8
```

A partir da versão 2.x, esse *comentário mágico* (*magic comment*, como é chamado) não é mais necessário se o *encoding* for UTF-8

Podemos verificar o *encoding* de uma String:

```
1 "eustáquio".encoding
2 => #<Encoding:UTF-8>
```

Podemos definir o *encoding* de uma String:

```
1 "eustáquio".encode "iso-8859-1"  
2 => "eust\xE1quio"  
3 "eustáquio".encode("iso-8859-1").encoding  
4 => #<Encoding:ISO-8859-1>
```



Novidade em Ruby 2.0

Temos agora o método `b` que converte uma `String` para a sua representação “binária”, ou seja, em ASCII:

```
1 "eustáquio".b  
2 => "eust\xC3\xA1quio"  
3 "eustáquio".b.encoding  
4 => #<Encoding:ASCII-8BIT>
```

Váriaveis são referências na memória

Em Ruby, os valores são transmitidos por referência, podendo verificar isso com `Strings`, constatando que as variáveis realmente armazenam referências na memória. Vamos notar que, se criarmos uma variável apontando para uma `String`, criamos outra apontando para a primeira (ou seja, para o mesmo local na memória) e se alterarmos a primeira, comportamento semelhante é notado na segunda variável:

```
1 nick = "TaQ"  
2 => "TaQ"  
3 other_nick = nick  
4 => "TaQ"  
5 nick[0] = "S"  
6 => "S"  
7 other_nick  
8 => "SaQ"
```

Para evitarmos que esse comportamento aconteça e realmente obter dois objetos distintos, podemos utilizar o método `dup`:

```
1 nick = "TaQ"
2 => "TaQ"
3 other_nick = nick.dup
4 => "TaQ"
5 nick[0] = "S"
6 => "S"
7 nick
8 => "SaQ"
9 other_nick
10 => "TaQ"
```

Congelando objetos

Se, por acaso quisermos que um objeto não seja modificado, podemos utilizar o método `freeze`:

```
1 nick = "TaQ"
2 => "TaQ"
3 nick.freeze
4 => "TaQ"
5 nick[0] = "S"
6 RuntimeError: can't modify frozen string
```

Não temos um método `unfreeze`, mas podemos gerar uma cópia do nosso objeto “congelado” com `dup`, e assim fazer modificações nessa nova cópia:

```
1 nick = "TaQ"
2 => "TaQ"
3 nick.freeze
4 => "TaQ"
5 new_nick = nick.dup
6 => "TaQ"
7 new_nick[0] = "S"
8 => "SaQ"
```

Alguns métodos e truques com Strings:

```
1 str = "tente"
2 str["nt"] = "st" => "teste"
3 str.sizea => 5
4 str.upcase => "TESTE"
5 str.upcase.downcase => "teste"
6 str.sub("t", "d") => "deste"
7 str.gsub("t", "d") => "desde"
8 str.capitalize => "Desde"
9 str.reverse => "etset"
10 str.split("t") => ["", "es", "e"]
11 str.scan("t") => ["t", "t"]
12 str.scan(/^t/) => ["t"]
13 str.scan(/.*/) => ["t", "e", "s", "t", "e"]
```

Símbolos

Símbolos, antes de mais nada, são instâncias da classe `Symbol`. Podemos pensar em um símbolo como uma marca, um nome, onde o que importa não é o que contém a sua instância, mas o seu nome.

Símbolos podem se parecer com um jeito engraçado de `Strings`, mas devemos pensar em símbolos como significado e não como conteúdo. Quando escrevemos “azul”, podemos pensar como um conjunto de letras, mas quando escrevemos `:azul`, podemos pensar em uma marca, uma referência para alguma coisa.

Símbolos também compartilham o mesmo `object_id`, em qualquer ponto do sistema:

```
1 :teste.class => Symbol
2 :teste.object_id
3 => 263928
4 :teste.object_id
5 => 263928
```

Como pudemos ver, as duas referências para os símbolos compartilham o mesmo objeto, enquanto que foram alocados dois objetos para as `Strings`. Uma boa economia de memória com apenas uma ressalva: símbolos não são objetos candidatos a limpeza automática pelo *garbage collector*, ou seja, se você alocar muitos, mas muitos símbolos no seu sistema, você poderá experimentar um nada agradável esgotamento de memória que com certeza não vai trazer coisas boas para a sua aplicação, ao contrário de `Strings`, que são alocadas mas liberadas quando não estão sendo mais utilizadas.

Outra vantagem de símbolos é a sua comparação. Para comparar o conteúdo de duas `Strings`, temos que percorrer os caracteres um a um e com símbolos podemos comparar os seus `object_ids` que sempre serão os mesmos, ou seja, uma comparação $O(1)$ (onde o tempo para completar é sempre constante e o mesmo e não depende do tamanho da entrada).

Imaginem o tanto que economizamos usando tal tipo de operação!

Expressões regulares

Outra coisa muito útil em Ruby é o suporte para expressões regulares (*regexps*). Elas podem ser facilmente criadas das seguintes maneiras:

```
1 regex1 = /^[0-9]/
2 => /^[0-9]/
3 regex2 = Regexp.new("[0-9]")
4 => /^[0-9]/
5 regex3 = %r{^[0-9]}
6 => /^[0-9]/
```

Para fazermos testes com as expressões regulares, podemos utilizar os operadores `=~` (“igual o tiozinho quem vos escreve”) que indica se a expressão “casou” e `!~` que indica se a expressão não “casou”, por exemplo:

```
1 "1 teste" =~ regex1
2 => 0
3 "1 teste" =~ regex2
4 => 0
5 "1 teste" =~ regex3
6 => 0
7 "outro teste" !~ regex1
8 => true
9 "outro teste" !~ regex2
10 => true
11 "outro teste" !~ regex3
12 => true
13 "1 teste" !~ regex1
14 => false
15 "1 teste" !~ regex2
16 => false
17 "1 teste" !~ regex3
18 => false
```

No caso das expressões que “casaram”, foi retornada a posição da *String* onde houve correspondência. Podemos fazer truques bem legais com expressões regulares e *Strings*, como por exemplo, dividir a nossa *String* através de uma expressão regular:

```
1 "o rato roeu a roupa do rei de Roma".scan(/r[a-z]+/i)
2 => ["rato", "roeu", "roupa", "rei", "Roma"]
```

Fica uma dica que podemos utilizar alguns modificadores no final da expressão regular, no caso acima, `/i` indica que a expressão não será *case sensitive*, ou seja, levará em conta caracteres em maiúsculo ou minúsculo.

Grupos

Podemos utilizar grupos nas expressões regulares, utilizando (e) para delimitar o grupo, e \$<número> para verificar onde o grupo “casou”:

```
1 "Alberto Roberto" =~ /(\w+)( )(\w+)/
2 => 0
3 $1
4 => "Alberto"
5 $2
6 => " "
7 $3
8 => "Roberto"
```

Também podemos utilizar \<número> para fazer alguma operação com os resultados da expressão regular assim:

```
1 "Alberto Roberto".sub(/(\w+)( )(\w+)/, '\3 \1')
2 => "Roberto Alberto"
```



Novidade em Ruby 2.0

Onigmo vai ser o novo *engine* de expressões regulares da versão 2.0. Ela parece ser bem baseada em Perl e aqui tem vários recursos que podem estar presentes nesse *engine*. Como exemplo de novidades, podemos utilizar esse aqui:

```
1 regexp = /^( [A-Z] )?[a-z]+(?(1)[A-Z] | [a-z])$/
2 p regexp =~ "foo"      #=> 0
3 p regexp =~ "foO"      #=> nil
4 p regexp =~ "FoO"      #=> 0
```

Ali é declarado o seguinte: (?(cond)yes|no) (reparem no primeiro ? e em |, que funcionam como o operador ternário ? e :), onde se cond for atendida, é avaliado yes, senão, no, por isso que foo, iniciando e terminando com caracteres minúsculos, casa com no, foO com maiúsculo no final não casa com nada e Foo casa com yes.

Grupos nomeados

A partir da versão 1.9, podemos usar **grupos nomeados** em nossas expressões regulares, como por exemplo:

```
1 matcher = /(?!<objeto>\w{5})(.*)((?!<cidade>\w{4})$)/.match("o rato roeu a roupa \
2 do rei de Roma")
3 matcher[:objeto] => "roupa"
4 matcher[:cidade] => "Roma"
```

Arrays

Arrays podemos definir como objetos que contém coleções de referências para outros objetos. Vamos ver um Array simples com números:

```
1 array = [1,2,3,4,5] => [1, 2, 3, 4, 5]
```

Em Ruby os Arrays podem conter tipos de dados diferentes também, como esse onde misturamos inteiros, flutuantes e Strings:

```
1 array = [1,2.3,"oi"] => [1, 2.3, "oi"]
```



Dica

Para criarmos Arrays de Strings o método convencional é:

```
1 array = ["um", "dois", "tres", "quatro"]
2 => ["um", "dois", "tres", "quatro"]
```

mas temos um atalho que nos permite economizar digitação com as aspas, que é o %w e pode ser utilizado da seguinte maneira:

```
1 array = %w(um dois tres quatro)
2 => ["um", "dois", "tres", "quatro"]
```

e em Ruby 2.x, podemos utilizar %i para criar um Array de símbolos:

```
1 %i(um dois tres quatro)
2 => [:um, :dois, :tres, :quatro]
```

Podemos também criar Arrays com tamanho inicial pré-definido utilizando o tamanho na criação do objeto:

```
1 array = Array.new(5)
2 => [nil, nil, nil, nil, nil]
```

Para indicar qual valor ser utilizado ao invés de nil nos elementos do Array criado com tamanho definido, podemos usar:

```
1 array = Array.new(5,0)
2 => [0, 0, 0, 0, 0]
```

Vamos verificar um efeito interessante, criando um Array com tamanho de 5 e algumas Strings como o valor de preenchimento:

```
1 array = Array.new(5, "oi")
2 => ["oi", "oi", "oi", "oi", "oi"]
```

Foi criado um Array com 5 elementos, mas são todos os mesmos elementos. Duvidam? Olhem só:

```
1 array[0].upcase!
2 => "OI"
3 array
4 => ["OI", "OI", "OI", "OI", "OI"]
```

Foi aplicado um método destrutivo (que alteram o próprio objeto da referência, não retornando uma cópia dele no primeiro elemento do Array, que alterou *todos os outros elementos*, pois são *o mesmo objeto*). Para evitarmos isso, podemos utilizar um bloco (daqui a pouco mais sobre blocos!) para criar o Array:

```
1 array = Array.new(5) { "oi" }
2 => ["oi", "oi", "oi", "oi", "oi"]
3 array[0].upcase!
4 => "OI"
5 array
6 => ["OI", "oi", "oi", "oi", "oi"]
```

Pudemos ver que agora são objetos distintos.

Aqui temos nosso primeiro uso para blocos de código, onde o bloco foi passado para o construtor do Array, que cria elementos até o número que especificamos transmitindo o valor do índice (ou seja, 0, 1, 2, 3 e 4) para o bloco.

Os Arrays tem uma característica interessante que vários outros objetos de Ruby tem: eles são **iteradores**, ou seja, objetos que permitem percorrer uma coleção de valores, pois incluem o módulo (hein? mais adiante falaremos sobre módulos!) `Enumerable`, que inclui essa facilidade.

Como parâmetro para o método que vai percorrer a coleção, vamos passar um bloco de código e vamos ver na prática como que funciona isso. Dos métodos mais comuns para percorrer uma coleção, temos `each`, que significa “cada”, e que pode ser lido “para cada elemento da coleção do meu objeto, execute esse bloco de código”, dessa maneira:

```

1 array.each {|numero| puts "O Array tem o numero "+numero.to_s }
2 O Array tem o numero 1
3 O Array tem o numero 2
4 O Array tem o numero 3
5 O Array tem o numero 4

```

Ou seja, para cada elemento do Array, foi executado o bloco, atenção aqui, passando o elemento corrente como parâmetro, recebido pelo bloco pela sintaxe |<parâmetro>|. Podemos ver que as instruções do nosso bloco, que no caso só tem uma linha (e foi usada a convenção de { e }), foram executadas com o valor recebido como parâmetro.



Dica

Temos um atalho em Ruby que nos permite economizar conversões dentro de Strings. Ao invés de usarmos `to_s` como mostrado acima, podemos utilizar o que é conhecido como interpolador de expressão com a sintaxe `#{objeto}`, onde tudo dentro das chaves vai ser transformado em `String` acessando o seu método `to_s`. Ou seja, poderíamos ter escrito o código do bloco como:

```
{|numero| puts "O Array tem o numero #{numero}"}
```

Podemos pegar sub-arrays utilizando o formato `[início..fim]` ou o método `take`:

```

1 a = %w(john paul george ringo)
2 => ["john", "paul", "george", "ringo"]
3 a[0..1]
4 => ["john", "paul"]
5 a[1..2]
6 => ["paul", "george"]
7 a[1..3]
8 => ["paul", "george", "ringo"]
9 a[0]
10 => "john"
11 a[-1]
12 => "ringo"
13 a.first
14 => "john"
15 a.last
16 => "ringo"
17 a.take(2) => ["john", "paul"]

```

Reparem no pequeno truque de usar `-1` para pegar o último elemento, o que pode ficar bem mais claro utilizando o método `last` (e `first` para o primeiro elemento).

Agora que vimos como um iterador funciona, podemos exercitar alguns outros logo depois de conhecer mais alguns outros tipos.

Para adicionar elementos em um Array, podemos utilizar o método `push` ou o `<<` (lembram desse, nas Strings), desse modo:

```
1 a = %w(john paul george ringo)
2 a.push("stu")
3 => ["john", "paul", "george", "ringo", "stu"]
4 a << "george martin"
5 => ["john", "paul", "george", "ringo", "stu", "george martin"]
```

Duck Typing

Pudemos ver que o operador/método `<<` funciona de maneira similar em `Strings` e `Arrays`, e isso é um comportamento que chamamos de **Duck Typing**, baseado no **duck test**, de **James Whitcomb Riley**, que diz o seguinte:

“Se parece com um pato, nada como um pato, e faz barulho como um pato, então provavelmente é um pato”.

Isso nos diz que, ao contrário de linguagens de tipagem estática, onde o tipo do objeto é verificado em tempo de compilação, em Ruby nos interessa se um objeto é capaz de exibir algum comportamento esperado, não o tipo dele.

Se você quer fazer uma omelete, não importa que animal que está botando o ovo (galinha, pata, avestruz, Tiranossauro Rex, etc), desde que você tenha um jeito/método para botar o ovo.

“Ei, mas como vou saber se um determinado objeto tem um determinado método?” Isso é fácil de verificar utilizando o método `respond_to?`:

```
1 String.new.respond_to?(:<<)
2 => true
3 Array.new.respond_to?(:<<)
4 => true
```

“Ei, mas eu realmente preciso saber se o objeto em questão é do tipo que eu quero. O método `<<` é suportado por `Arrays`, `Strings`, `Fixnums` mas tem comportamento diferente nesses últimos!”. Nesse caso, você pode verificar o tipo do objeto utilizando `kind_of?`:

```
1 String.new.kind_of?(String)
2 => true
3 1.kind_of?(Fixnum)
4 => true
5 1.kind_of?(Numeric)
6 => true
7 1.kind_of?(Bignum)
8 => false
```

Ranges

Ranges são intervalos que podemos definir incluindo ou não o último valor referenciado. Vamos exemplificar isso com o uso de iteradores, dessa maneira:

```

1 range1 = (0..10)
2 => 0..10
3 range2 = (0...10)
4 => 0...10
5 range1.each {|valor| print "#{valor} "}
6 0 1 2 3 4 5 6 7 8 9 10 => 0..10
7 range2.each {|valor| print "#{valor} "}
8 0 1 2 3 4 5 6 7 8 9 => 0...10

```

Como podemos ver, as Ranges são declaradas com um valor inicial e um valor final, separadas por dois ou três pontos, que definem se o valor final vai constar ou não no intervalo.



Dica

Para se lembrar qual das duas opções que incluem o valor, se lembre que nesse caso **menos é mais**, ou seja, **com dois pontos temos mais valores**.

Um truque legal é que podemos criar Ranges com Strings:

```

1 ("a".. "z").each {|valor| print "#{valor} "}
2 a b c d e f g h i j k l m n o p q r s t u v w x y z
3 => "a".. "z"
4 ("ab".. "az").each {|valor| print "#{valor} "}
5 ab ac ad ae af ag ah ai aj ak al am an ao ap aq ar as at au av aw ax ay az
6 => "ab".. "az"

```

Outro bem legal é converter uma Range em um Array:

```

1 ("a".. "z").to_a
2 => ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
3 , "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]

```

Hashes

As Hashes são, digamos, Arrays indexados, com chaves e valores, que podem ser quaisquer tipos de objetos, como por exemplo:

```

1 hash = {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra", :ri\
2 ngo=>"bateria"}
3 => {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra", :ring\
4 o=>"bateria"}

```

No Ruby 1.9.x as Hashes mantêm a ordem dos elementos do jeito que foram criadas, porém em algumas versões do Ruby 1.8.x essa ordem é aleatória. Depois de declaradas, podemos buscar os seus valores através de suas chaves:

```
1 hash[:paul]
2 => "baixo e voz"
3 hash[:ringo]
4 => "bateria"
```

Vamos ver um exemplo de como podemos armazenar diversos tipos tanto nas chaves como nos valores de uma Hash:

```
1 hash = {"fixnum"=>1, :float=>1.23, 1=>"um"}
2 => {1=>"um", :float=>1.23, "fixnum"=>1}
3 hash["fixnum"]
4 => 1
5 hash[:float]
6 => 1.23
7 hash[1]
8 => "um"
```

Podemos criar Hashes com valores default:

```
1 hash = Hash.new(0)
2 => {}
3 hash[:um]
4 => 0
5 hash[:dois]
6 => 0
```

Nesse caso, quando o valor da chave ainda não teve nada atribuído e é requisitado, é retornado o valor default que especificamos em `new`, que foi 0. Vamos testar com outro valor:

```
1 hash = Hash.new(Time.now)
2 => {}
3 hash[:um]
4 => Tue Jun 05 23:53:22 -0300 2011
5 hash[:dois]
6 => Tue Jun 05 23:53:22 -0300 2011
```

No caso acima, passei `Time.now` no método `new` da Hash, e toda vez que tentei acessar um dos valores que ainda não foram atribuídos, sempre foi retornado o valor da data e hora de quando inicializei a Hash. Para que esse valor possa ser gerado dinamicamente, podemos passar um bloco para o método `new`:

```

1 hash = Hash.new { Time.now }
2 => {}
3 hash[:um]
4 => 2008-12-31 11:31:28 -0200
5 hash[:dois]
6 => 2008-12-31 11:31:32 -0200
7 hash[:tres]
8 => 2008-12-31 11:31:36 -0200

```

Hashes são bastante utilizadas como parâmetros de vários métodos do Rails.



Dica

A partir do Ruby 1.9, podemos criar Hashes dessa maneira:

```

1 hash = {john: "guitarra e voz", paul: "baixo e voz", george: "guitarra", ringo\
2 : "bateria"}
3 => {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra", :rin\
4 go=>"bateria"}

```

Reparem que o operador “rocket”(⇒) sumiu.

Blocos de código

Um conceito interessante do Ruby são blocos de código (similares ou sendo a mesma coisa em certos sentidos que funções anônimas, *closures*, *lambdas* etc). Vamos ir aprendendo mais coisas sobre eles no decorrer do curso, na prática, mas podemos adiantar que blocos de código são uma das grande sacadas de Ruby e são muito poderosos quando utilizados com iteradores.

Por convenção os blocos com uma linha devem ser delimitados por { e } e com mais de uma linha com do ... end (duende??), mas nada lhe impede de fazer do jeito que mais lhe agrada. Como exemplo de blocos, temos:

```

1 puts {"Oi, mundo"}

e

1 do
2   puts "Oi, mundo"
3   puts "Aqui tem mais linhas!"
4 end

```

Esses blocos podem ser enviados para métodos e executados pelos iteradores de várias classes. Imaginem como pequenos pedaços de código que podem ser manipulados e enviados entre os métodos dos objetos (tendo eles próprios, comportamento de métodos).

Conversões de tipos

Agora que vimos os tipos mais comuns, podemos destacar que temos algumas métodos de conversão entre eles, que nos permitem transformar um tipo (mas não o mesmo objeto, será gerado um novo) em outro. Alguns dos métodos:

```
1 Fixnum para Float
2 1.to_f => 1.0
3
4 Fixnum para String
5 1.to_s => "1"
6
7 String para Fixnum
8 "1".to_i => 1
9
10 String para flutuante
11 "1".to_f => 1.0
12
13 String para símbolo
14 "azul".to_sym => :azul
15
16 Array para String
17 [1,2,3,4,5].to_s => "12345"
18
19 Array para String, com delimitador
20 [1,2,3,4,5].join(", ") => "1,2,3,4,5"
21
22 Range para Array
23 (0..10).to_a => [0,1,2,3,4,5,6,7,8,9,10]
24
25 Hash para Array
26 {:john=>"guitarra e voz"}.to_a=>[[:john,"guitarra e voz"]]
```

Conversões de bases

De inteiro para binário:

```
1 2.to_s(2)
2 => "10"
```

De binário para inteiro:

```
1 "10".to_i(2)
2 => 2
3 0b10.to_i
4 => 2
```

De inteiro para hexadecimal:

```
1 10.to_s(16)
2 => "a"
```

De hexadecimal para inteiro:

```
1 0xa.to_i
2 => 10
```

Tratamento de exceções

Exceções nos permitem “cercar” erros que acontecem no nosso programa (afinal, ninguém é perfeito, não é mesmo?) em um objeto que depois pode ser analisado e tomadas as devidas providências ao invés de deixar o erro explodir dentro do nosso código levando à resultados indesejados. Vamos gerar um erro de propósito para testar isso.

Lembram-se que Ruby tem uma tipagem forte, onde não podemos misturar os tipos de objetos? Vamos tentar misturar:

```
1 begin
2   numero = 1
3   string = "oi"
4   numero + string
5 rescue StandardError => exception
6   puts "Ocorreu um erro: #{exception}"
7 end
```

Rodando o programa, temos:

```
1 Ocorreu um erro: String can't be coerced into Fixnum
```

O programa gerou uma exceção no código contido entre `begin` e `rescue` interceptando o tipo de erro tratado pela exceção do tipo `StandardError`, em um objeto que foi transmitido para `rescue`, através da variável `exception`, onde pudemos verificar informações sobre o erro, imprimindo-o como uma `String`.

Se não quisermos especificar o tipo de exceção a ser tratada, podemos omitir o tipo, e verificar a classe da exceção gerada dessa maneira:

```
1 begin
2   numero = 1
3   string = "oi"
4   numero + string
5 rescue => exception
6   puts "Ocorreu um erro do tipo #{exception.class}: #{exception}"
7 end
```

Rodando o programa, temos:

```
1 Ocorreu um erro do tipo TypeError: String can't be coerced into Fixnum
```

Podemos utilizar `ensure` como um bloco para ser executado depois de todos os `rescues`:

```
1 begin
2   numero = 1
3   string = "oi"
4   numero + string
5 rescue => exception
6   puts "Ocorreu um erro do tipo #{exception.class}: #{exception}"
7 ensure
8   puts "Lascou tudo."
9 end
10 puts "Fim de programa."
```

Rodando o programa:

```
1 Ocorreu um erro do tipo TypeError: String can't be coerced into Fixnum
2 Lascou tudo.
3 Fim de programa.
```

Isso é particularmente interessante se houver algum problema dentro de algum bloco de `rescue`:

```
1 begin
2   numero = 1
3   string = "oi"
4   numero + string
5 rescue => exception
6   puts "Ocorreu um erro do tipo #{exception.class}: #{exception}"
7   puts msg
8 ensure
9   puts "Lascou tudo."
10 end
11 puts "Fim de programa."
```

Rodando o programa:

```
1 Ocorreu um erro do tipo TypeError: String can't be coerced into Fixnum
2 Lascou tudo.
3 exc1.rb:7: undefined local variable or method 'msg' for main:Object (NameError\
4 r)
```

Podemos ver que foi gerada uma nova exceção dentro do bloco do `rescue` e apesar do comando final com a mensagem “Fim de programa” não ter sido impressa pois a exceção “jogou” o fluxo de processamento para fora, o bloco do `ensure` foi executado.

Se por acaso desejarmos tentar executar o bloco que deu problema novamente, podemos utilizar `retry`:

```
1 numero1 = 1
2 numero2 = "dois"
3 begin
4   puts numero1 + numero2
5 rescue => exception
6   puts "Ops, problemas aqui (#{exception.class}), vou tentar de novo."
7   numero2 = 2
8   retry
9 end
```

Rodando o programa:

```
1 Ops, problemas aqui (TypeError), vou tentar de novo.
2 3
```

Se desejarmos ter acesso a `backtrace` (a lista hierárquica das linhas dos programas onde o erro ocorreu), podemos utilizar:

```
1 numero1 = 1
2 numero2 = "dois"
3 begin
4   puts numero1 + numero2
5 rescue => exception
6   puts "Ops, problemas aqui (#{exception.class}), vou tentar de novo."
7   puts exception.backtrace
8   numero2 = 2
9   retry
10 end
```

Rodando o programa, nesse caso chamado `exc1.rb`, vai nos retornar:

```
1 Ops, problemas aqui (TypeError), vou tentar de novo.
2 exc1.rb:4:in '+':
3 exc1.rb:4
4 3
```

Disparando exceções

Podemos disparar exceções utilizando `raise`:

```
1 numero1 = 1
2 numero2 = 1
3 begin
4   puts numero1 + numero2
5   raise Exception.new("esperava 3") if numero1+numero2!=3
6 rescue => exception
7   puts "Ops, problemas aqui (#{exception.class}), vou tentar de novo."
8 end
```

Descobrimo a exceção anterior

Podemos descobrir qual foi a exceção que foi disparada anteriormente utilizando `cause`, que nos dá acesso as *nested exceptions* (a partir da versão 2.1.):

```
1 begin
2   begin
3     raise 'foo'
4   rescue Exception => foo
5     raise 'bar'
6   end
7 rescue Exception => bar
8   puts "a causa de #{bar} foi #{bar.cause}"
9 end
```

Para versões anteriores, dá para utilizar ou a gem `cause` ⁹.

Criando nossas próprias exceções

Se por acaso quisermos criar nossas próprias classes de exceções, é muito fácil, basta criá-las herdando de `StandardError`. Vamos criar uma que vamos disparar se um nome for digitado errado, `NameNotEqual`:

⁹<https://github.com/ConradIrwin/cause>

```

1  # encoding: utf-8
2
3  class NameNotEqual < StandardError
4      def initialize(current,expected)
5          super "você digitou um nome inválido (#{current})! era esperado #{exp\
6  ected}."
7      end
8  end
9
10 begin
11     correct = "eustaquio"
12     puts "digite o meu nome: "
13     name = gets.chomp
14     raise NameNotEqual.new(name,correct) if name!=correct
15     puts "digitou correto!"
16 rescue NameNotEqual => e
17     puts e
18 end

```

Rodando o programa e digitando qualquer coisa diferente de “eustaquio”:

```

1  $ ruby customexceptions.rb
2  digite o meu nome:
3  barizon
4  você digitou um nome inválido (barizon)! era esperado eustaquio.

```

Utilizando catch e throw

Também podemos utilizar `catch` e `throw` para terminar o processamento quando nada mais é necessário, indicando através de um `Symbol` para onde o controle do código deve ser transferido (opcionalmente com um valor), indicado com `catch`, usando `throw`:

```

1  # encoding: utf-8
2
3  def get_input
4      puts "Digite algo (número termina):"
5      resp = gets
6      throw :end_of_response, resp if resp.chomp =~ /\d+$/
7      resp
8  end
9
10 num = catch(:end_of_response) do
11     while true
12         get_input
13     end

```

```
14 end
15 puts "Terminado com: #{num}"
```

Rodando:

```
1 Digite algo (número termina):
2 oi
3 Digite algo (número termina):
4 123
5 Terminado com: 123
```

Estruturas de controle

Condicionais

if

É importante notar que tudo em Ruby acaba no fim – end – e vamos ver isso acontecendo bastante com nossas estruturas de controle. Vamos começar vendo nosso velho amigo `if`:

```
1 i = 10
2 => 10
3 if i == 10
4   puts "i igual 10"
5 else
6   puts "i diferente de 10"
7 end
8 i igual 10
```

Uma coisa bem interessante em Ruby é que podemos escrever isso de uma forma que podemos “ler” o código, se, como no caso do próximo exemplo, estivermos interessados apenas em imprimir a mensagem no caso do teste do ‘if’ ser verdadeiro:

```
1 puts "i igual 10" if i==10
2 i igual 10
```

Isso é chamado de **modificador de estrutura**.

Também temos mais um nível de teste no `if`, o `elsif`:

```
1 i = 10
2 => 10
3 if i > 10
4   puts "maior que 10"
5 elsif i == 10
6   puts "igual a 10"
7 else
8   puts "menor que 10"
9 end
10 igual a 10
```

Podemos capturar a saída do teste diretamente apontando uma variável para ele:

```
1 i = 10
2 => 10
3 result =
4 if i > 10
5     "maior que 10"
6 elsif i == 10
7     "igual a 10"
8 else
9     "menor que 10"
10 end
11 => "igual a 10"
12 result
13 => "igual a 10"
```

unless

O `unless` é a forma negativa do `if`, e como qualquer teste negativo, pode trazer alguma confusão no jeito de pensar sobre eles. Particularmente gosto de evitar testes negativos quando pode-se fazer um bom teste positivo.

Vamos fazer um teste imaginando uma daquelas cadeiras de boteco e alguns sujeitos mais avantajados (em peso, seus mentes sujas):

```
1 peso = 150
2 => 150
3 puts "pode sentar aqui" unless peso > 100
4 => nil
5 peso = 100
6 => 100
7 puts "pode sentar aqui" unless peso > 100
8 pode sentar aqui
```

Dá para lermos o comando como “diga ao sujeito que ele pode sentar aqui a menos que o peso dele for maior que 100 quilos”. Talvez um teste mais limpo seria:

```
1 peso = 150
2 => 150
3 puts "pode sentar aqui" if peso <= 100
4 => nil
5 peso = 100
6 => 100
7 puts "pode sentar aqui" if peso <= 100
8 pode sentar aqui
```

Ler “diga ao sujeito que ele pode sentar aqui se o peso for menor ou igual a 100” talvez seja um jeito mais claro de fazer o teste, mas fica a critério de cada um e do melhor uso.

case

Podemos utilizar o case para fazer algumas comparações interessantes. Vamos ver como testar com Ranges:

```
1 i = 10
2 => 10
3 case i
4 when 0..5
5   puts "entre 0 e 5"
6 when 6..10
7   puts "entre 6 e 10"
8 else
9   puts "hein?"
10 end
11 entre 6 e 10
```

No caso do case (redundância detectada na frase), a primeira coisa que ele compara é o tipo do objeto, nos permitindo fazer testes como:

```
1 i = 10
2 => 10
3 case i
4 when Fixnum
5   puts "Número!"
6 when String
7   puts "String!"
8 else
9   puts "hein???"
10 end
11 Número!
```

Para provar que esse teste tem precedência, podemos fazer:

```
1 i = 10
2 => 10
3 case i
4 when Fixnum
5   puts "Número!"
6 when (0..100)
7   puts "entre 0 e 100"
8 end
9 Número!
```

A estrutura `case` compara os valores de forma invertida, como no exemplo acima, `Fixnum ===` e não `i === Fixnum`, não utilizando o operador `==` e sim o operador `===`, que é implementado das seguintes formas:

Para **módulos** e **classes** (que vamos ver mais à frente), é comparado se o valor é uma instância do módulo ou classe ou de um de seus descendentes. No nosso exemplo, `i` é uma instância de `Fixnum`. Por exemplo:

```
1 Fixnum === 1
2 => true
3 Fixnum === 1.23
4 => false
```

Para expressões regulares, é comparado se o valor “casou” com a expressão:

```
1 /[0-9]/ === "123"
2 => true
3 /[0-9]/ === "abc"
4 => false
```

Para `Ranges`, é testado se o valor se inclui nos valores da `Range` (como no método `include?`):

```
1 (0..10) === 1
2 => true
3 (0..10) === 100
4 => false
```

Loops

Antes de vermos os *loops*, vamos deixar anotado que temos algumas maneiras de interagir dentro de um loop:

1. **break** - sai do loop
2. **next** - vai para a próxima iteração
3. **return** - sai do loop e do método onde o loop está contido
4. **redo** - repete o loop do início, sem reavaliar a condição ou pegar o próximo elemento

Vamos ver exemplos disso logo na primeira estrutura a ser estudada, o `while`.



Dica

A partir desse ponto vamos utilizar um editor de texto para escrevermos nossos exemplos, usando o `irb` somente para testes rápidos com pouco código. Você pode utilizar o editor de texto que quiser, desde que seja um editor mas não um processador de textos. Não vá utilizar o Microsoft Word © para fazer os seus programas, use um editor como o Vim. Edite o seu código, salve em um arquivo com a extensão `.rb` e execute da seguinte forma (onde `$` é o *prompt* do terminal):

```
$ruby meuarquivo.rb
```

while

Faça enquanto:

```
1 i = 0
2 while i < 5
3   puts i
4   i += 1
5 end
6
7 $ruby while.rb
8 0
9 1
10 2
11 3
12 4
```

for

O for pode ser utilizado junto com um iterador para capturar todos os seus objetos e enviá-los para o loop (que nada mais é do que um bloco de código):

```
1 for i in (0..5)
2   puts i
3 end
4
5 $ruby for.rb
6 0
7 1
8 2
9 3
10 4
11 5
```

Vamos aproveitar que é um *loop* bem simples e utilizar os comandos para interagir mostrados acima (mesmo que os exemplos pareçam as coisas mais inúteis e sem sentido do mundo - mas é para efeitos didáticos, gente!), menos o `return` onde precisaríamos de um método e ainda não chegamos lá. Vamos testar primeiro o `break`:

```
1 for i in (0..5)
2   break if i==3
3   puts i
4 end
5
6 $ruby for.rb
7 0
8 1
9 2
```

Agora o next:

```
1 for i in (0..5)
2   next if i==3
3   puts i
4 end
5
6 $ruby for.rb
7 0
8 1
9 2
10 4
11 5
```

Agora o redo:

```
1 for i in (0..5)
2   redo if i==3
3   puts i
4 end
5
6 $ruby for.rb
7 0
8 1
9 2
10 for.rb:2: Interrupt
11 from for.rb:1:in 'each'
12 from for.rb:1
```

Se não interrompermos com Ctrl+C, esse código vai ficar funcionando para sempre, pois o redo avaliou o *loop* novamente mas sem ir para o próximo elemento do iterador.

until

O “faça até que” pode ser utilizado dessa maneira:

```
1 i = 0
2 until i==5
3   puts i
4   i += 1
5 end
6
7 $ruby until.rb
8 0
9 1
10 2
11 3
12 4
```



Dica

Não temos os operadores ++ e -- em Ruby. Utilize += e -=.

Operadores lógicos

Temos operadores lógicos em Ruby em duas formas: !, &&, || e not, and, or. Eles se diferenciam pela precedência: os primeiros tem precedência mais alta que os últimos sobre os operadores de atribuição. Exemplificando:

```
1 a = 1
2 => 1
3 b = 2
4 => 2
5 c = a && b
6 => 2
7 c
8 => 2
9 d = a and b
10 => 2
11 d
12 => 1
```

A variável `c` recebeu o resultado correto de `a && b`, enquanto que `d` recebeu a atribuição do valor de `a` e seria a mesma coisa escrito como `(d = a) and b`. O operador avalia o valor mais à direita somente se o valor mais a esquerda não for falso. É a chamada operação de “curto-circuito”.

Outro exemplo de “curto-circuito” é o operador `||=` (chamado de “ou igual” ou “*pipe* duplo igual”, que funciona da seguinte maneira:

```

1 a ||= 10
2 => 10
3 a
4 => 10
5 a ||= 20
6 => 10
7 a
8 => 10

```

O que ocorre ali é o seguinte: é atribuído o valor à variável **apenas se o valor dela for false ou nil, do contrário, o valor é mantido**. Essa é uma forma de curto-circuito pois seria a mesma coisa que:

```
1 a || a = 10
```

que no final das contas retorna a se o valor for diferente de false e nil ou, do contrário, faz a atribuição do valor para a variável. Seria basicamente

```
1 a || (a = 10)
```



Dica

Temos um método chamado `defined?` que testa se a referência que passamos para ele existe. Se existir, ele retorna uma descrição do que é ou `nil` se não existir. Exemplo:

```

1 a, b, c = (0..2).to_a
2 => [0, 1, 2]
3 defined? a
4 => "local-variable"
5 defined? b
6 => "local-variable"
7 defined? String
8 => "constant"
9 defined? 1.next
10 => "method"

```



Desafio 1

Declare duas variáveis, `x` e `y`, com respectivamente 1 e 2 como valores, com apenas uma linha. Agora inverta os seus valores também com apenas uma linha de código. O que vamos fazer aqui é uma **atribuição em paralelo**. Resposta no final do livro!

Procs e lambdas

Procs são blocos de código que podem ser associados à uma variável, dessa maneira:

```
1 vezes3 = Proc.new {|valor| valor*3}
2 => #<Proc:0xb7d959c4@(irb):1>
3 vezes3.call(3)
4 => 9
5 vezes3.call(4)
6 => 12
7 vezes3.call(5)
8 => 15
```

Comportamento similar pode ser alcançada usando `lambda`:

```
1 vezes5 = lambda {|valor| valor*5}
2 => #<Proc:0xb7d791d4@(irb):5>
3 vezes5.call(5)
4 => 25
5 vezes5.call(6)
6 => 30
7 vezes5.call(7)
8 => 35
```

Pudemos ver que precisamos executar `call` para chamar a `Proc`, mas também podemos utilizar o atalho `[]`:

```
1 vezes5[8]
2 => 40
```

E também o atalho `.`, menos comum:

```
1 vezes5.(5)
2 => 25
```

Podemos utilizar uma `Proc` como um bloco, mas para isso precisamos convertê-la usando `&`:

```
1 (1..5).map &vezes5
2 => [5, 10, 15, 20, 25]
```



Dica

Fica um “gancho” aqui sobre o fato de `Procs` serem *closures*, ou seja, códigos que criam uma cópia do seu ambiente. Quando estudarmos métodos vamos ver um exemplo prático sobre isso.

Importante notar duas diferenças entre `Procs` e `lambdas`:

A primeira diferença, é a *verificação de argumentos*. Em `lambdas` a verificação é feita e gera uma exceção:

```

1  pnew = Proc.new {|x, y| puts x + y}
2  => #<Proc:0x8fdaf7c@(irb):7>
3  lamb = lambda {|x, y| puts x + y}
4  => #<Proc:0x8fd7aac@(irb):8 (lambda)>
5  pnew.call(2,4,11)
6  => nil
7  lamb.call(2,4,11)
8  ArgumentError: wrong number of arguments (3 for 2)

```

A segunda diferença é o jeito que elas retornam. O retorno de uma Proc retorna de dentro de onde ela está, como nesse caso:

```

1  def testando_proc
2    p = Proc.new { return "Bum!" }
3    p.call
4    "Nunca imprime isso."
5  end
6  puts testando_proc
7
8  $ ruby code/procret.rb
9  Bum!

```

Enquanto que em uma lambda, retorna para onde foi chamada:

```

1  def testando_lambda
2    l = lambda { return "Oi!" }
3    l.call
4    "Imprime isso."
5  end
6  puts testando_lambda
7  $ ruby code/lambret.rb
8  Imprime isso.

```

A partir do Ruby 1.9, temos suporte à sintaxe “*stabby proc*”:

```

1  p = -> x,y { x* y }
2  puts p.call(2,3)
3
4  $ ruby code/stabproc.rb
5  6

```

E também ao método `curry`, que decompõe uma lambda em uma série de outras lambdas. Por exemplo, podemos ter uma lambda que faça multiplicação:

```

1 mult = lambda {|n1,n2| n1*n2}
2 => #<Proc:0x8fef1fc@(irb):13 (lambda)>
3 mult.(2,3)
4 => 6

```

E podemos utilizar o método `curry` no final e ter o seguinte resultado:

```

1 mult = lambda {|n1,n2| n1*n2}.curry
2 => #<Proc:0x8ffe4e0 (lambda)>
3 mult.(2).(3)
4 => 6

```

Reparem que o método `call` (na forma de `.()`) foi chamado **duas vezes**, primeiro com 2 e depois com 3, pois o método `curry` inseriu uma `lambda` dentro da outra, como se fosse:

```

1 multi = lambda {|x| lambda {|y| x*y}}
2 => #<Proc:0x901756c@(irb):23 (lambda)>
3 multi.(2).(3)
4 => 6

```

Isso pode ser útil quando você deseja criar uma `lambda` a partir de outra, deixando um dos parâmetros fixo, como por exemplo:

```

1 mult = lambda {|n1,n2| n1*n2}.curry
2 => #<Proc:0x901dd40 (lambda)>
3 dobro = mult.(2)
4 => #<Proc:0x901c058 (lambda)>
5 triplo = mult.(3)
6 => #<Proc:0x9026904 (lambda)>
7 dobro.(8)
8 => 16
9 triplo.(9)
10 => 27

```

Iteradores

Agora que conhecemos os tipos básicos de Ruby, podemos focar nossa atenção em uma característica bem interessante deles: muitos, senão todos, tem coleções ou características que podem ser percorridas por métodos iteradores.

Um iterador percorre uma determinada coleção, que o envia o valor corrente, executando algum determinado procedimento, que em Ruby é enviado como um bloco de código e contém o módulo (hein?) `Enumerable`, que dá as funcionalidades de que ele precisa.

Dos métodos mais comuns para percorrer uma coleção, temos `each`, que significa “cada”, e que pode ser lido “para cada elemento da coleção do meu objeto, execute esse bloco de código”, dessa maneira:

```
1 [1,2,3,4,5].each { |e| puts "o array contem o numero #{e}" }
```

Rodando o programa:

```
1 ruby code/it1.rb
2 array contem o numero 1
3 array contem o numero 2
4 array contem o numero 3
5 array contem o numero 4
6 array contem o numero 5
```

Ou seja, para cada elemento do Array, foi executado o bloco - atenção aqui - passando o elemento corrente como parâmetro, recebido pelo bloco pela sintaxe `|<parâmetro>|`. Podemos ver que as instruções do nosso bloco, que no caso só tem uma linha (e foi usada a convenção de `{ e }`), foram executadas com o valor recebido como parâmetro.

Esse mesmo código pode ser otimizado e refatorado para ficar mais de acordo com a sua finalidade. Não precisamos de um *loop* de 1 até 5? A maneira mais adequada seria criar uma Range com esse intervalo e executar nosso iterador nela:

```
1 (1..5).each { |e| puts "a range contem o numero #{e}" }
2
3 ruby code/it2.rb
4 range contem o numero 1
5 range contem o numero 2
6 range contem o numero 3
7 range contem o numero 4
8 range contem o numero 5
```

Inclusive, podemos também utilizar `times` em um Fixnum, que se comporta como uma coleção nesse caso, que começa em 0:

```
1 5.times { |e| puts "numero #{e}" }
2
3 ruby code/it2b.rb
4 numero 0
5 numero 1
6 numero 2
7 numero 3
8 numero 4
```

Um Array só faria sentido nesse caso se os seus elementos não seguissem uma ordem lógica que pode ser expressa em um intervalo de uma Range! Quaisquer sequências que podem ser representadas fazem sentido em usar uma Range. Se por acaso quiséssemos uma lista de números de 1 até 21, em intervalos de 3, podemos utilizar:

```
1 (1..21).step(2).each { |e| puts "numero #{e}" }
2
3 $ ruby code/it3.rb
4 numero 1
5 numero 3
6 numero 5
7 numero 7
8 numero 9
9 numero 11
10 numero 13
11 numero 15
12 numero 17
13 numero 19
14 numero 21
```

Em Rails utilizamos bastante a estrutura `for <objeto> in <coleção>`, da seguinte forma:

```
1 col = %w(uma lista de Strings para mostrar o for)
2 for str in col
3   puts str
4 end
5 $ ruby code/it4.rb
6 uma
7 lista
8 de
9 Strings
10 para
11 mostrar
12 o
13 for
```

Selecionando elementos

Vamos supor que queremos selecionar alguns elementos que atendam alguma condição nos nossos objetos, por exemplo, selecionar apenas os números pares de uma coleção:

```
1 (1..10).select { |e| e.even? }
2 => [2, 4, 6, 8, 10]
```

Vamos testar com uma Hash:

```
1 {1=>"um",2=>"dois",3=>"tres"}.select { |chave,valor| valor.length>2}
2 => {2=>"dois", 3=>"tres"}
```



Novidade em Ruby 2.0

Aproveitando o gancho do método `select`, temos agora no Ruby 2.0 o conceito de *lazy evaluation*. Reparem no método `lazy` antes do `select`:

```

1  natural_numbers = Enumerator.new do |yielder|
2    number = 1
3    loop do
4      yielder.yield number
5      number += 1
6    end
7  end
8
9  p natural_numbers.lazy.select { |n| n.odd? }.take(5).to_a
10 # => [1, 3, 5, 7, 9]
```

Se não utilizássemos `lazy`, íamos precisar de um CTRL+C, pois o conjunto de números naturais é infinito, e a seleção nunca pararia para que fossem pegos os 5 elementos.

Selecionando os elementos que não atendem uma condição

O contrário da operação acima pode ser feito com `reject`:

```

1  (0..10).reject {|valor| valor.even?}
2  => [1, 3, 5, 7, 9]
```

Nada que a condição alterada do `select` também não faça.

Processando e alterando os elementos

Vamos alterar os elementos do objeto com o método `map`:

```

1  (0..10).map {|valor| valor*2}
2  => [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
3  %w(um dois tres quatro cinco seis sete oito nove dez).map {|valor| "numero #{\
4  valor}"}
5  => ["numero um", "numero dois", "numero tres", "numero quatro",
6  "numero cinco", "numero seis", "numero sete", "numero oito", "numero nove",
7  "numero dez"]
8
9  {1=>"um",2=>"dois",3=>"tres"}.map {|chave,valor| "numero #{valor}" }
10 => ["numero um", "numero dois", "numero tres"]
```

Detectando condição em todos os elementos

Vamos supor que desejamos detectar se todos os elementos da coleção atendem uma determinada condição com o método `all?`:

```
1 (0..10).all? {|valor| valor>1}
2 => false
3 (0..10).all? {|valor| valor>0}
4 => false
```

Detectando se algum elemento atende uma condição

Vamos testar se algum elemento atende uma determinada condição com o método `any?`:

```
1 (0..10).any? {|valor| valor==3}
2 => true
3 (0..10).any? {|valor| valor==30}
4 => false
```

Nesse caso específico, poderíamos ter escrito dessa forma também:

```
1 (0..10).include?(3)
2 => true
3 (0..10).include?(30)
4 => false
```

Apesar da facilidade com um teste simples, o método `any?` é muito prático no caso de procurarmos, por exemplo, um determinado objeto com um determinado valor de retorno em algum de seus métodos.

Detectar e retornar o primeiro elemento que atende uma condição

Se além de detectar quisermos retornar o elemento que atende à uma condição, podemos utilizar o método `detect?`:

```
1 (0..10).detect {|valor| valor>0 && valor%4==0}
2 => 4
```

Detectando os valores máximo e mínimo

Podemos usar `max` e `min` para isso:

```
1 (0..10).max
2 => 10
3 (0..10).min
4 => 0
```

É interessante notar que podemos passar um bloco onde serão comparados os valores para teste através do operador `<=>` (conhecido por “navinha”):

```

1 %w(joao maria antonio).max {|elemento1,elemento2| elemento1.length <=> elemen\
2 to2.length}
3 => "antonio"
4 %w(joao maria antonio).min {|elemento1,elemento2| elemento1.length <=> elemen\
5 to2.length}
6 => "joao"

```



Dica

O operador `<=>` compara o objeto da esquerda com o objeto da direita e retorna -1 se o objeto à esquerda for menor, 0 se for igual e 1 se for maior do que o da direita:

```

1 1 <=> 2 => -1
2 1 <=> 1 => 0
3 1 <=> -1 => 1

```

Olhem que interessante comparando valores de Hashes:

```

1 {:joao=>33,:maria=>30,:antonio=>25}.max {|elemento1,elemento2| elemento1[1] <\
2 => elemento2[1]}
3 => [:joao, 33]
4 {:joao=>33,:maria=>30,:antonio=>25}.min {|elemento1,elemento2| elemento1[1] <\
5 => elemento2[1]}
6 => [:antonio, 25]

```



Desafio 2

Tem uma mágica de conversão escondida ali. Você consegue descobrir qual é?

Acumulando os elementos

Podemos acumular os elementos com `inject`, onde vão ser passados um valor acumulador e o valor corrente pego do iterador. Se desejarmos saber qual é a soma de todos os valores da nossa Range:

```

1 (0..10).inject {|soma,valor| soma+valor}
2 => 55

```

Podemos passar também um valor inicial:

```
1 (0..10).inject(100) {|soma,valor| soma+valor}
2 => 155
```

E também podemos passar o método que desejamos utilizar para combinação como um símbolo:

```
1 (0..10).inject(:+)
2 => 55
3 (0..10).inject(100, :+)
4 => 155
```

Dividir a coleção em dois Arrays obedecendo uma condição

Vamos separar os números pares dos ímpares usando `partition`:

```
1 (0..10).partition {|valor| valor.even?}
2 => [[0, 2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
```

Percorrendo os elementos com os índices

Vamos ver onde cada elemento se encontra com `each_with_index`:

```
1 (0..10).each_with_index {|item,indice| puts "#{item} indice #{indice}" }
2 0 indice 0
3 1 indice 1
4 2 indice 2
5 3 indice 3
6 4 indice 4
7 5 indice 5
8 6 indice 6
9 7 indice 7
10 8 indice 8
11 9 indice 9
12 10 indice 10
```

Ordenando uma coleção

Vamos ordenar um Array de Strings usando `sort`:

```
1 %w(joao maria antonio).sort
2 => ["antonio", "joao", "maria"]
```

Podemos ordenar de acordo com algum critério específico, passando um bloco e usando `sort_by`:

```
1 %w(antonio maria joao).sort_by {|nome| nome.length}
2 => ["joao", "maria", "antonio"]
```

Combinando elementos

Podemos combinar elementos com o método zip:

```
1 (1..10).zip((11..20))
2 => [[1, 11], [2, 12], [3, 13], [4, 14], [5, 15], [6, 16], [7, 17], [8, 18], [9, 19], [10, 20]]
3
4 (1..10).zip((11..20),(21..30))
5 => [[1, 11, 21], [2, 12, 22], [3, 13, 23], [4, 14, 24], [5, 15, 25], [6, 16, 26], [7, 17, 27], [8, 18, 28], [9, 19, 29], [10, 20, 30]]
6
```

Também podemos usar combination:

```
1 a = %w(john paul george ringo)
2 => ["john", "paul", "george", "ringo"]
3 a.combination(2)
4 => #<Enumerable::Enumerator:0xb7d711a0>
5 a.combination(2).to_a
6 => [{"john", "paul"}, {"john", "george"}, {"john", "ringo"}, {"paul", "george"}, {"paul", "ringo"}, {"george", "ringo"}]
7
8
9 a.combination(2) {|comb| puts "combinando #{comb[0]} com #{comb[1]}"}
10 combinando john com paul
11 combinando john com george
12 combinando john com ringo
13 combinando paul com george
14 combinando paul com ringo
15 combinando george com ringo
```

Ou permutation:

```
1 a = %w(john paul george ringo)
2 => ["john", "paul", "george", "ringo"]
3 a.permutation(2)
4 => #<Enumerable::Enumerator:0xb7ce41c4>
5 a.permutation(2).to_a
6 => [{"john", "paul"}, {"john", "george"}, {"john", "ringo"}, {"paul", "john"}, {"paul", "george"}, {"paul", "ringo"}, {"george", "john"}, {"george", "paul"}, {"george", "ringo"}, {"ringo", "john"}, {"ringo", "paul"}, {"ringo", "george"}]
7
8
9
10
```

```

11 a.permutation(2) {|comb| puts "combinando #{comb[0]} com #{comb[1]}" }
12 combinando john com paul
13 combinando john com george
14 combinando john com ringo
15 combinando paul com john
16 combinando paul com george
17 combinando paul com ringo
18 combinando george com john
19 combinando george com paul
20 combinando george com ringo
21 combinando ringo com john
22 combinando ringo com paul
23 combinando ringo com george

```

Ou product:

```

1 beatles = %w(john paul george ringo)
2 => ["john", "paul", "george", "ringo"]
3 stooges = %w(moe larry curly shemp)
4 => ["moe", "larry", "curly", "shemp"]
5
6 beatles.product(stooges)
7 => [ ["john", "moe"], ["john", "larry"], ["john", "curly"], ["john", "shemp"], \
8     ["paul", "moe"], ["paul", "larry"], ["paul", "curly"], ["paul", "shemp"], ["\
9     george", "moe"], ["george", "larry"], ["george", "curly"], ["george", "shemp"\
10    ], ["ringo", "moe"], ["ringo", "larry"], ["ringo", "curly"],

```

[“ringo”, “shemp”]]

Percorrendo valores para cima e para baixo

Podemos usar upto, downto e step:

```

1 1.upto(5) {|num| print num, " "}
2 1 2 3 4 5 => 1
3 5.downto(1) {|num| print num, " "}
4 5 4 3 2 1 => 5
5 1.step(10,2) {|num| print num, " "}
6 1 3 5 7 9 => 1

```

Inspecionando no encadeamento de métodos

Um método bem útil para o caso de precisarmos inspecionar ou registrar o conteúdo de algum objeto durante algum encadeamento de iteradores é o método tap. Vamos supor que você tem o seguinte código:

```
1 (0..10).select {|num| num.even?}.map {|num| num*2}
2 => [0, 4, 8, 12, 16, 20]
```

Isso nada mais faz do que separar os números pares e multiplicá-los por 2, mas imaginemos que a coleção inicial não é formada por números e sim por objetos da nossa tabela de funcionários onde vamos selecionar somente algumas pessoas que atendem determinadas condições (usando o `select`) e reajustar o seu salário baseado em várias regras complexas (o `map`), e algum problema está ocorrendo na seleção.

O jeito convencional é criar uma variável temporária armazenando o conteúdo retornado pelo `select` e a imprimirmos, executando o `map` logo em seguida. Ou podemos fazer assim:

```
1 (0..10).select {|num| num.even?}.tap{|col| p col}.map {|num| num*2}
2 [0, 2, 4, 6, 8, 10]
3 => [0, 4, 8, 12, 16, 20]
```

Isso nos mostra o conteúdo antes de ser enviado para o próximo método encadeado.

Métodos

Podemos definir métodos facilmente em Ruby, usando `def`, terminando (como sempre) com `end`:

```
1 def diga_oi
2   puts "Oi!"
3 end
4 diga_oi
5 => "Oi!"
```

Executando esse código, será impresso `Oi!`. Já podemos reparar que os parênteses não são obrigatórios para chamar um método em Ruby.

Retornando valores

Podemos retornar valores de métodos com ou **sem** o uso de `return`. Quando não utilizamos `return`, o que ocorre é que a **última expressão avaliada é retornada**, como no exemplo:

```
1 def vezes(p1,p2)
2   p1*p2
3 end
4 puts vezes(2,3)
5 => 6
```

No caso, foi avaliado por último `p1*p2`, o que nos dá o resultado esperado. Também podemos retornar mais de um resultado, que na verdade é apenas um objeto, sendo ele complexo ou não, dando a impressão que são vários, como no exemplo que vimos atribuição em paralelo.

Vamos construir um método que retorna cinco múltiplos de um determinado número:

```
1 def cinco_multiplos(numero)
2   (1..5).map {|valor| valor*numero}
3 end
4 v1,v2,v3,v4,v5 = cinco_multiplos(5)
5 puts "#{v1},#{v2},#{v3},#{v4},#{v5}"
6 => 5,10,15,20,25
```

Enviando valores

Antes de mais nada, fica a discussão sobre a convenção sobre o que são parâmetros e o que são argumentos, convencionando-se à:

Parâmetros são as variáveis situadas na assinatura de um método; Argumentos são os valores atribuídos aos parâmetros

Vimos acima um exemplo simples de passagem de valores para um método, vamos ver outro agora:

```
1 def vezes(n1,n2)
2   n1*n2
3 end
4 puts vezes(3,4)
5 => 12
```

Podemos contar quantos parâmetros um método recebe usando arity:

```
1 def vezes(n1,n2)
2   n1*n2
3 end
4 puts vezes(3,4)
5 puts "o metodo recebe #{method(:vezes).arity} parametros"
```

Métodos também podem receber parâmetros *default*, como por exemplo:

```
1 def oi(nome="Forasteiro")
2   puts "Oi, #{nome}!"
3 end
4 oi("TaQ")
5 => Oi, TaQ!
6 oi
7 => Oi, Forasteiro!
```

E também valores variáveis, bastando declarar o nosso método como recebendo um parâmetro com o operador `splat` (asterisco, `*`) antes do nome do parâmetro:

```

1 def varios(*valores)
2   valores.each {|valor| puts "valor=#{valor}"}
3   puts "-"*25
4 end
5 varios(1)
6 valor=1
7 -----
8 varios(1,2)
9 valor=1
10 valor=2
11 -----
12 varios(1,2,3)
13 valor=1
14 valor=2
15 valor=3
16 -----

```

O operador `splat` pode parecer meio estranho, mas ele nada mais faz, na definição do método, do que concentrar todos os valores recebidos em um Array, como pudemos ver acima.

Quando usamos o `splat` na frente do nome de uma variável que se comporta como uma coleção, ele “explode” os seus valores, retornando os elementos individuais:

```

1 array = %w(um dois tres)
2 => ["um", "dois", "tres"]
3 p *array
4 "um"
5 "dois"
6 "tres"
7 => nil
8 hash = {:um=>1, :dois=>2, :tres=>3}
9 => {:tres=>3, :um=>1, :dois=>2}
10 p *hash
11 [:tres, 3]
12 [:um, 1]
13 [:dois, 2]
14 => nil

```

Enviando e processando blocos e Procs

Como vimos com iteradores, podemos passar um bloco para um método, e para o executarmos dentro do método, usamos `yield`:

```
1 def executa_bloco(valor)
2   yield(valor)
3 end
4
5 executa_bloco(2) {|valor| puts valor*valor}
6 => 4
7 executa_bloco(3) {|valor| puts valor*valor}
8 => 9
9 executa_bloco(4) {|valor| puts valor*valor}
10 => 16
```

Podemos usar `block_given?` para detectar se um bloco foi passado para o método:

```
1 def executa_bloco(valor)
2   yield(valor) if block_given?
3 end
4 executa_bloco(2) {|valor| puts valor*valor}
5 => 4
6 executa_bloco(3)
7 executa_bloco(4) {|valor| puts valor*valor}
8 16
```

Podemos também converter um bloco em uma Proc especificando o nome do último parâmetro com `&` no começo:

```
1 def executa_bloco(valor, &proc)
2   puts proc.call(valor)
3 end
4 executa_bloco(2) {|valor| valor*valor}
5 => 4
```

Valores são transmitidos por referência

Como recebemos referências do objeto nos métodos, quaisquer alterações que fizermos dentro do método refletirão fora, como já vimos um pouco acima quando falando sobre variáveis. Vamos comprovar:

```
1 def altera(valor)
2   valor.upcase!
3 end
4 string = "Oi, mundo!"
5 altera(string)
6 puts string
7 => "OI, MUNDO!"
```

Interceptando exceções direto no método

Uma praticidade grande é usarmos o corpo do método para capturarmos uma exceção, sem precisar abrir um bloco com begin e end:

```
1 def soma(valor1,valor2)
2   valor1+valor2
3 rescue
4   nil
5 end
6 puts soma(1,2) => 3
7 puts soma(1,:um) => nil
```

Também podemos utilizar os caracteres ! e ? no final dos nomes dos nossos métodos. Por convenção, métodos com ! no final são métodos destrutivos e com ? no final são métodos para testar algo:

```
1 def revup!(str)
2   str.reverse!.upcase!
3 end
4 str = "teste"
5 puts str.object_id
6 revup!(str)
7 puts str
8 puts str.object_id
9
10 $ ruby code/destructive.rb
11 74439960
12 ETSET
13 74439960
14
15 def ok?(obj)
16   !obj.nil?
17 end
18
19 puts ok?(1)
20 puts ok?("um")
```

```
21 puts ok?(:um)
22 puts ok?(nil)
23
24 $ ruby code/testmeth.rb
25 true
26 true
27 true
28 false
```

Podemos simular argumentos nomeados usando uma Hash:

```
1 def test(args)
2   one = args[:one]
3   two = args[:two]
4   puts "one: #{one} two: #{two}"
5 end
6
7 test(one: 1, two: 2)
8 test(two: 2, one: 1)
9
10 $ ruby code/named.rb
11 one: 1 two: 2
12 one: 1 two: 2
```



Novidade em Ruby 2.0

A partir de Ruby 2.0 já temos suporte para argumentos nomeados:

```
1 def foo(str: "foo", num: 123456)
2   [str, num]
3 end
4
5 p foo(str: 'buz', num: 9)
6 => ['buz', 9]
7 p foo(str: 'bar')
8 => ['bar', 123456]
9 p foo(num: 123)
10 => ['foo', 123]
11 p foo
12 => ['foo', 123456]
13 p foo(bar: 'buz') # => ArgumentError
```

Também podemos capturar um método como se fosse uma Proc:

```

1 class Teste
2   def teste(qtde)
3     qtde.times { puts "teste!" }
4   end
5 end
6 t = Teste.new
7 m = t.method(:teste)
8 p m
9 m.(3)
10 p m.to_proc
11
12 $ ruby code/capture.rb
13 #<Method: Teste#teste>
14 teste!
15 teste!
16 teste!
17 #<Proc:0x8d3c4b4 (lambda)>

```

Como podemos ver, o resultado é um objeto do tipo Method, mas que pode ser convertido em uma Proc usando o método to_proc.

E agora um método de nome totalmente diferente usando o suporte para encodings do Ruby a partir das versões 1.9.x:

```

1 # encoding: utf-8
2 module Enumerable
3   def Σ
4     self.inject {|memo, val| memo += val}
5   end
6 end
7
8 puts [1,2,3].Σ
9 puts (0..3).Σ
10
11 $ ruby code/encodingmeth.rb
12 6
13 6

```

Uau! Para quem quiser inserir esses caracteres malucos no Vim, consulte o help dos *digraphs* com :help digraphs. Esse do exemplo é feito usando, no modo de inserção, CTRL+K +Z.

Classes e objetos

Como bastante coisas em Ruby são objetos, vamos aprender a criar os nossos. Vamos fazer uma classe chamada `Carro`, com algumas propriedades:

```
1 class Carro
2   def initialize(marca, modelo, cor, tanque)
3     @marca = marca
4     @modelo = modelo
5     @cor = cor
6     @tanque = tanque
7   end
8 end
9 corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
10 p corsa
11 puts corsa
12
13 $ ruby code/carro1.rb
14 #<Carro:0x894c674 @marca=:chevrolet, @modelo=:corsa, @cor=:preto, @tanque=50>
15 #<Carro:0x894c674>
```

Para criarmos uma classe, usamos a palavra-chave `class`, seguida pelo nome da classe.

Segundo as convenções de Ruby, nos nomes dos métodos deve-se usar letras minúsculas separando as palavras com um sublinhado (`_`), porém nos nomes das classes é utilizado *camel case*, da mesma maneira que em Java, com maiúsculas separando duas ou mais palavras no nome da classe. Temos então classes com nomes como `MinhaClasse`, `MeuTeste`, `CarroPersonalizado`.

As propriedades do nosso objeto são armazenadas no que chamamos variáveis de instância, que são quaisquer variáveis dentro do objeto cujo nome se inicia com `@`. Se fizermos referência para alguma que ainda não foi criada, ela será. Podemos inicializar várias dessas variáveis dentro do método `initialize`, que é o **construtor** do nosso objeto, chamado **após** o método `new`, que aloca espaço na memória para o objeto sendo criado.

Não temos métodos destrutores em Ruby, mas podemos associar uma `Proc` para ser chamada em uma instância de objeto cada vez que ela for limpa pelo *garbage collector*. Vamos verificar isso criando o arquivo `destructor.rb`:

```
1 string = "Oi, mundo!"
2 ObjectSpace.define_finalizer(string, lambda {|id| puts "Estou terminando o obj\
3 eto #{id}"})
```

E agora rodando, o que vai fazer com que todos os objetos sejam destruídos no final:

```
1 $ ruby destructor.rb
2 Estou terminando o objeto 78268620
```



Dica

Podemos pegar um objeto pelo seu `object_id`:

```
1 s = "oi"
2 => "oi"
3 i = s.object_id
4 => 80832250
5 puts ObjectSpace._id2ref(i)
6 oi
```



Desafio 3

Crie mais algumas variáveis/referências como no exemplo acima, associando uma Proc com o finalizer do objeto. Repare que talvez algumas não estejam exibindo a mensagem. Porque?

Pudemos ver acima que usando `puts` para verificar o nosso objeto, foi mostrada somente a referência dele na memória. Vamos fazer um método novo na classe para mostrar as informações de uma maneira mais bonita. Lembra-se que em conversões utilizamos um método chamado `to_s`, que converte o objeto em uma `String`? Vamos criar um para a nossa classe:

```
1 class Carro
2   def initialize(marca, modelo, cor, tanque)
3     @marca = marca
4     @modelo = modelo
5     @cor = cor
6     @tanque = tanque
7   end
8
9   def to_s
10    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
11  end
12 end
13
14 corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
15 p corsa
16 puts corsa
```

Vamos ver o comportamento nas versões 1.8.x:

```
1 $ rvm 1.8.7
2 $ ruby code/carro2.rb
3 #<Carro:0xb75be6b0 @cor=:preto, @modelo=:corsa, @marca=:chevrolet, @tanque=50>
4 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
```

E agora nas versões 1.9.x:

```
1 $ rvm 1.9.3
2 $ ruby code/carro2.rb
3 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
4 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
```

E agora nas versões 2.x:

```
1 $ rvm 2.0.0
2 $ ruby code/carro2.rb
3 #<Carro:0x85808b0 @marca=:chevrolet, @modelo=:corsa, @cor=:preto, @tanque=50>
4 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
```



Sobrescrever o método `to_s` não deveria afetar o `inspect`. O pessoal discutiu muito isso e nas versões 2.x foi restaurado o comportamento antes das 1.9.x, como visto acima.



Ruby tem alguns métodos que podem confundir um pouco, parecidos com `to_s` e `to_i`, que são `to_str` e `to_int`. Enquanto `to_s` e `to_i` efetivamente fazem uma conversão de tipos, `to_str` e `to_int` indicam que os objetos podem ser representados como uma `String` e um `Fixnum`, respectivamente. Ou seja: `to_s` significa que o objeto **tem** representação como `String`, `to_str` significa que o objeto **é** uma representação de `String`.



Todo método chamado sem um receiver explícito será executado em `self`, que especifica o próprio objeto ou classe corrente.

Vimos como criar as propriedades do nosso objeto através das variáveis de instância, mas como podemos acessá-las? Isso vai nos dar um erro:

```
1 class Carro
2   def initialize(marca,modelo,cor,tanque)
3     @marca = marca
4     @modelo = modelo
5     @cor = cor
6     @tanque = tanque
7   end
8
9   def to_s
10    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
11  end
12 end
13
14 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
15 puts corsa.marca
16
17 $ ruby code/carro3.rb
18 code/carro3.rb:14:in '<main>': undefined method 'marca' for Marca:chevrolet
19 Modelo:corsa Cor:preto Tanque:50:Carro (NoMethodError)
```

Essas variáveis são **privadas** do objeto, e não podem ser lidas sem um método de acesso. Podemos resolver isso usando `attr_reader`:

```
1 class Carro
2   attr_reader :marca, :modelo, :cor, :tanque
3
4   def initialize(marca,modelo,cor,tanque)
5     @marca = marca
6     @modelo = modelo
7     @cor = cor
8     @tanque = tanque
9   end
10
11   def to_s
12    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
13  end
14 end
15
16 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
17 puts corsa.marca
18
19 $ ruby code/carro4.rb
20 chevrolet
```

Nesse caso, criamos atributos de leitura, que nos permitem a leitura da propriedade. Se precisarmos de algum atributo de escrita, para trocarmos a cor do carro, por exemplo, podemos usar:

```
1 class Carro
2   attr_reader :marca, :modelo, :cor, :tanque
3   attr_writer :cor
4
5   def initialize(marca,modelo,cor,tanque)
6     @marca = marca
7     @modelo = modelo
8     @cor = cor
9     @tanque = tanque
10  end
11
12  def to_s
13    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
14  end
15 end
16
17 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
18 corsa.cor = :branco
19 puts corsa
20
21 $ ruby code/carro5.rb
22 Marca:chevrolet Modelo:corsa Cor:branco Tanque:50
```

Podemos até encurtar isso mais ainda criando direto um atributo de escrita e leitura com attr_accessor:

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque
3   attr_accessor :cor
4
5   def initialize(marca,modelo,cor,tanque)
6     @marca = marca
7     @modelo = modelo
8     @cor = cor
9     @tanque = tanque
10  end
11
12  def to_s
13    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
14  end
15 end
16
17 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
18 corsa.cor = :branco
19 puts corsa
20
```

```
21 $ ruby code/carro6.rb
22 Marca:chevrolet Modelo:corsa Cor:branco Tanque:50
```



Dica

Se precisarmos de objetos com atributos com escrita e leitura, podemos usar duas formas bem rápidas para criarmos nossos objetos. Uma é usando Struct:

```
1 Carro = Struct.new(:marca, :modelo, :tanque, :cor)
2 corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
3 p corsa
4 => #<struct Carro marca=:chevrolet, modelo=:corsa, tanque=:preto, cor=50>
```

Outra é mais flexível ainda, usando OpenStruct, onde os atributos são criados na hora que precisamos deles:

```
1 require "ostruct"
2 carro = OpenStruct.new
3 carro.marca = :chevrolet
4 carro.modelo = :corsa
5 carro.cor = :preto
6 carro.tanque = 50
7 p carro
8 => #<OpenStruct tanque=50, modelo=:corsa, cor=:preto, marca=:chevrolet>
```

Também podemos criar atributos virtuais, que nada mais são do que métodos que agem como se fossem atributos do objeto. Vamos supor que precisamos de uma medida como galões, que equivalem a 3,785 litros, para o tanque do carro. Poderíamos fazer:

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque
3   attr_accessor :cor
4
5   def initialize(marca, modelo, cor, tanque)
6     @marca = marca
7     @modelo = modelo
8     @cor = cor
9     @tanque = tanque
10  end
11
12  def to_s
13    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
14  end
15
```

```
16     def galoes
17         @tanque / 3.785
18     end
19 end
20
21 corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
22 corsa.cor = :branco
23 puts corsa.galoes
24
25 $ ruby code/carro7.rb
26 13.21003963011889
```

Classes abertas

Uma diferença de Ruby com várias outras linguagens é que as suas classes, mesmo as definidas por padrão e base na linguagem, são abertas, ou seja, podemos alterá-las depois que as declararmos. Por exemplo:

```
1 # encoding: utf-8
2 class Carro
3     attr_reader :marca, :modelo, :tanque
4     attr_accessor :cor
5
6     def initialize(marca, modelo, cor, tanque)
7         @marca = marca
8         @modelo = modelo
9         @cor = cor
10        @tanque = tanque
11    end
12
13    def to_s
14        "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
15    end
16 end
17
18 corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
19
20 class Carro
21     def novo_metodo
22         puts "Novo método!"
23     end
24 end
25 corsa.novo_metodo
26
27 class Carro
```

```
28     remove_method :novo_metodo
29 end
30
31 corsa.novo_metodo
32 $ ruby code/carro8.rb
33 Novo método!
34 code/carro8.rb:30:in '<main>': undefined method 'novo_metodo' for
35 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50:Carro (NoMethodError)
```

Pude inserir e remover um método que é incorporado aos objetos que foram definidos sendo daquela classe e para os novos a serem criados também. Também pudemos remover o método, o que gerou a mensagem de erro.

Aliases

Se por acaso quisermos guardar uma cópia do método que vamos redefinir, podemos usar `alias_method` para dar outro nome para ele:

```
1  # encoding: utf-8
2  class Carro
3      attr_reader :marca, :modelo, :tanque
4      attr_accessor :cor
5
6      def initialize(marca,modelo,cor,tanque)
7          @marca = marca
8          @modelo = modelo
9          @cor = cor
10         @tanque = tanque
11     end
12
13     def to_s
14         "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
15     end
16 end
17
18 class Carro
19     alias_method :to_s_old, :to_s
20     def to_s
21         "Esse é um novo jeito de mostrar isso: #{@to_s_old}"
22     end
23 end
24
25 carro = Carro.new(:chevrolet,:corsa,:preto,50)
26 puts carro
27 puts carro.to_s_old
```

```
28
29 $ ruby code/methalias.rb
30 Esse é um novo jeito de mostrar isso: Marca:chevrolet Modelo:corsa Cor:preto \
31 Tanque:50
32 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
```

Inserindo e removendo métodos

Podemos também inserir um método somente em uma determinada instância:

```
1 # encoding: utf-8
2 class Carro
3   attr_reader :marca, :modelo, :tanque
4   attr_accessor :cor
5
6   def initialize(marca, modelo, cor, tanque)
7     @marca = marca
8     @modelo = modelo
9     @cor = cor
10    @tanque = tanque
11  end
12
13  def to_s
14    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
15  end
16 end
17
18 corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
19 gol = Carro.new(:volks, :gol, :azul, 42)
20
21 class << corsa
22   def novo_metodo
23     puts "Novo método!"
24   end
25 end
26
27 corsa.novo_metodo
28 gol.novo_metodo
29
30 $ ruby code/insmethinst.rb
31 Novo método!
32 code/insmethinst.rb:28:in '<main>': undefined method 'novo_metodo' for
33 Marca:volks Modelo:gol Cor:azul Tanque:42:Carro (NoMethodError)
```

Podemos ver que no caso do corsa, o novo método foi adicionado, mas não no gol. O que aconteceu ali com o operador <<? Hora de algumas explicações sobre **metaclasses!**

Metaclasses

Todo objeto em Ruby tem uma hierarquia de ancestrais, que podem ser vistos utilizando `ancestors`, como:

```
1 class Teste
2 end
3 p String.ancestors
4 p Teste.ancestors
5
6 $ ruby ancestors.rb
7 [String, Comparable, Object, Kernel, BasicObject]
8 [Teste, Object, Kernel, BasicObject]
```

E cada objeto tem a sua **superclasse**:

```
1 class Teste
2 end
3
4 class OutroTeste < Teste
5 end
6
7 p String.superclass
8 p Teste.superclass
9 p OutroTeste.superclass
10
11 $ ruby superclasses.rb
12 Object
13 Object
14 Teste
```

Todos os objetos a partir das versões 1.9.x são derivados de `BasicObject`, que é o que chamamos de *Blank Slate*, que é um objeto que tem menos métodos que `Object`.

```
1 BasicObject.instance_methods
2 => [:, :equal?, :!, :!, :instance_eval, :instance_exec, :__send__]
```

O que ocorreu no exemplo da inserção do método na instância acima (quando utilizamos `<<`), é que o método foi inserido na **metaclass**, ou *eigenclass*, ou classe *singleton*, ou “classe fantasma” do objeto, que adiciona um novo elo na hierarquia dos ancestrais da classe da qual a instância pertence, ou seja, o método foi inserido antes da classe `Carro`. A procura do método (*method lookup*) se dá na *eigenclass* da instância, depois na hierarquia de ancestrais.



Em linguagens de tipagem estática, o compilador checa se o objeto *receiver* tem um método com o nome especificado. Isso é chamado checagem estática de tipos (*static type checking*), daí o nome da característica dessas linguagens.

Para isso ficar mais legal e prático, vamos ver como fazer dinamicamente, já começando a brincar com metaprogramação ¹⁰. Primeiro, com a classe:

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque
3   attr_accessor :cor
4
5   def initialize(marca, modelo, cor, tanque)
6     @marca = marca
7     @modelo = modelo
8     @cor = cor
9     @tanque = tanque
10  end
11
12  def to_s
13    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
14  end
15 end
16
17 corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
18 gol = Carro.new(:volks, :gol, :azul, 42)
19
20 Carro.send(:define_method, "multiplica_tanque") do |valor|
21   @tanque * valor
22 end
23
24 puts corsa.multiplica_tanque(2)
25 puts gol.multiplica_tanque(2)
26
27 $ ruby code/carro9.rb
28 100
29 84
```



Dica

Usamos `send` para acessar um método **privado** da classe.

Agora, com as instâncias:

¹⁰Metaprogramação é escrever código que manipula a linguagem em *runtime*.

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque
3   attr_accessor :cor
4
5   def initialize(marca,modelo,cor,tanque)
6     @marca = marca
7     @modelo = modelo
8     @cor = cor
9     @tanque = tanque
10  end
11
12  def to_s
13    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
14  end
15 end
16
17 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
18 gol = Carro.new(:volks,:gol ,:azul ,42)
19
20 (class << corsa; self; end).send(:define_method,"multiplica_tanque") do |valo\
21 r|
22   @tanque * valor
23 end
24
25 puts corsa.multiplica_tanque(2)
26 puts gol.multiplica_tanque(2)
27 100
28 code/carro10.rb:25:in '<main>': undefined method 'multiplica_tanque' for
29 Marca:volks Modelo:gol Cor:azul Tanque:42:Carro (NoMethodError)
```

Depois de ver tudo isso sobre inserção e remoção de métodos dinamicamente, vamos ver um truquezinho para criar um método “autodestrutivo”:

```
1 class Teste
2   def apenas_uma_vez
3     def self.apenas_uma_vez
4       raise Exception, "Esse metodo se destruiu!"
5     end
6     puts "Vou rodar apenas essa vez hein?"
7   end
8 end
9
10 teste = Teste.new
11 teste.apenas_uma_vez
12 teste.apenas_uma_vez
13
```

```

14 $ ruby code/autodestruct.rb
15 Vou rodar apenas essa vez hein?
16 code/autodestruct.rb:4:in `apenas_uma_vez': Esse metodo se destruiu!
17 (Exception) from code/autodestruct.rb:12:in `<main>'

```

Isso não é algo que se vê todo dia, yeah! :-)

Variáveis de classe

Também podemos ter variáveis de classes, que são variáveis que se encontram no **contexto da classe e não das instâncias dos objetos da classe**. Variáveis de classes tem o nome começado com @@ e devem ser inicializadas antes de serem usadas. Por exemplo:

```

1 class Carro
2   attr_reader :marca, :modelo, :tanque
3   attr_accessor :cor
4   @@qtde = 0
5
6   def initialize(marca,modelo,cor,tanque)
7     @marca = marca
8     @modelo = modelo
9     @cor = cor
10    @tanque = tanque
11    @@qtde += 1
12  end
13
14  def to_s
15    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
16  end
17
18  def qtde
19    @@qtde
20  end
21 end
22
23 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
24 gol = Carro.new(:volks,:gol,:azul ,42)
25 ferrari = Carro.new(:ferrari,:viper,:vermelho ,70)
26
27 puts ferrari.qtde
28
29 $ ruby code/classvar1.rb
30 3

```

Para que não precisemos acessar a variável através de uma instância, podemos criar um **método de classe**, utilizando `self`. antes do nome do método:

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque
3   attr_accessor :cor
4   @@qtde = 0
5
6   def initialize(marca,modelo,cor,tanque)
7     @marca = marca
8     @modelo = modelo
9     @cor = cor
10    @tanque = tanque
11    @@qtde += 1
12  end
13
14  def to_s
15    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
16  end
17
18  def self.qtde
19    @@qtde
20  end
21 end
22
23 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
24 gol = Carro.new(:volks ,:gol ,:azul ,42)
25 ferrari = Carro.new(:ferrari,:enzo ,:vermelho ,70)
26 puts Carro.qtde
27
28 $ ruby code/classvar2.rb
29 3
```

Os métodos de classe também podem ser chamados de **métodos estáticos**, em que não precisam de uma instância da classe para funcionar. Fazendo uma pequena comparação com variáveis e métodos estáticos em Java, no arquivo CarroEstatico.java:

```
1 public class CarroEstatico {
2   private static int qtde = 0;
3
4   public CarroEstatico() {
5     ++qtde;
6   }
7
8   public static int qtde() {
9     return qtde;
10  }
11
12  public static void main(String args[]) {
```

```

13     CarroEstatico[] carros = new CarroEstatico[10];
14     for(int i=0; i<carros.length; i++) {
15         carros[i] = new CarroEstatico();
16         System.out.println(CarroEstatico.qtde()+" carros");
17     }
18 }
19 }

```

Rodando o programa:

```

1 $ java CarroEstatico
2
3
4 1 carros
5 2 carros
6 3 carros
7 4 carros
8 5 carros
9 6 carros
10 7 carros
11 8 carros
12 9 carros
13 10 carros

```

Interfaces fluentes

O método `self` é particularmente interessante para desenvolvermos *interfaces fluentes*¹¹, que visa a escrita de código mais legível, geralmente implementada utilizando métodos encadeados, auto-referenciais no contexto (ou seja, sempre se referindo ao mesmo objeto) até que seja encontrado e retornado um contexto vazio. Poderíamos ter uma interface fluente bem básica para montar alguns comandos `select SQL` dessa forma:

```

1 class SQL
2     attr_reader :table, :conditions, :order
3     def from(table)
4         @table = table
5         self
6     end
7
8     def where(cond)
9         @conditions = cond
10        self
11    end

```

¹¹http://en.wikipedia.org/wiki/Fluent_interface

```
12
13   def order(order)
14     @order = order
15     self
16   end
17
18   def to_s
19     "select * from #{@table} where #{@conditions} order by #{@order}"
20   end
21 end
22
23 sql = SQL.new.from("carros").where("marca='Ford']").order("modelo")
24 puts sql
```

Rodando o programa:

```
$ ruby fluent.rb select * from carros where marca='Ford' order by modelo
```

Reparem que `self` sempre foi retornado em todos os métodos, automaticamente retornando o próprio objeto de onde o método seguinte do encadeamento foi chamado.

Variáveis de instância de classe

Um problema que acontece com as variáveis de classe utilizando `@@` é que elas não pertencem realmente às classes, e sim à hierarquias, podendo permear o código dessa maneira:

```
1 @@qtde = 10
2
3 class Carro
4   attr_reader :marca, :modelo, :tanque
5   attr_accessor :cor
6   @@qtde = 0
7   puts self
8
9   def initialize(marca,modelo,cor,tanque)
10     @marca = marca
11     @modelo = modelo
12     @cor = cor
13     @tanque = tanque
14     @@qtde += 1
15   end
16
17   def to_s
18     "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
19   end
20
```

```

21     def self.qtde
22         @@qtde
23     end
24 end
25
26 puts self
27 puts @@qtde
28
29 $ ruby code/classvar3.rb
30 Carro
31 main
32 0

```



Em Ruby 2.0, rodar esse programa nos retorna um *warning*:

```

1     $ ruby classvar3.rb
2     classvar3.rb:2: warning: class variable access from toplevel
3     Carro
4     main
5     classvar3.rb:28: warning: class variable access from toplevel
6     0

```

Está certo que esse não é um código comum de se ver, mas já dá para perceber algum estrago quando as variáveis @@ são utilizadas dessa maneira. Repararam que a @@qtde *externa* teve o seu valor atribuído como 0 **dentro** da classe?

Podemos prevenir isso usando **variáveis de instância de classe**:

```

1 class Carro
2     attr_reader :marca, :modelo, :tanque
3     attr_accessor :cor
4
5     class << self
6         attr_accessor :qtde
7     end
8     @qtde = 0
9
10    def initialize(marca,modelo,cor,tanque)
11        @marca = marca
12        @modelo = modelo
13        @cor = cor
14        @tanque = tanque
15        self.class.qtde += 1
16    end

```

```
17
18     def to_s
19         "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
20     end
21 end
22
23 corsa = Carro.new(:chevrolet,:corsa,:preto,50)
24 gol    = Carro.new(:volks ,:gol ,:azul ,42)
25 ferrari = Carro.new(:ferrari,:enzo ,:vermelho ,70)
26 puts Carro.qtde
27
28 $ ruby code/classvar4.rb
29 3
```

Vejam que a variável está na **instância da classe** (sim, classes tem uma instância “flutuando” por aí) e não em instâncias de objetos criados pela classe (os @) e nem são variáveis de classe (os @@).

Herança

Em Ruby, temos **herança única**, que significa que uma classe pode apenas ser criada herdando de apenas outra classe, reduzindo a complexidade do código. Como exemplo de alguma complexidade (pouca, nesse caso), vamos pegar de exemplo esse código em C++:

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Automovel {
6  public:
7      void ligar() {
8          cout << "ligando o automóvel\n";
9      }
10 };
11
12 class Radio {
13 public:
14     void ligar() {
15         cout << "ligando o rádio\n";
16     }
17 };
18
19 class Carro: public Automovel, public Radio {
20 public:
21     Carro() {}
```

```

22 };
23
24 int main() {
25     Carro carro;
26     carro.ligar();
27     return 0;
28 }

```

Se compilarmos esse código, vamos ter esse resultado:

```

1 $ g++ -g -o carro carro.cpp
2 carro.cpp: Na função 'int main()':
3 carro.cpp:26:10: erro: request for member 'ligar' is ambiguous
4 carro.cpp:14:9: erro: candidates are: void Radio::ligar()
5 carro.cpp:7:9: erro:          void Automovel::ligar()

```

Não foi possível resolver qual método `ligar` era para ser chamado. Para isso, temos que indicar explicitamente em qual das classes herdadas o método vai ser chamado, trocando

```
1 carro.ligar();
```

para

```
1 carro.Automovel::ligar();
```

que resulta em

```

1 $ g++ -g -o carro carro.cpp
2 $ ./carro
3 ligando o automóvel

```

Para fazermos a herança nas nossas classes em Ruby, é muito simples, é só utilizarmos `class` <nome da classe filha> < <nome da classe pai>:

```

1 class Carro
2     attr_reader :marca, :modelo, :tanque
3     attr_accessor :cor
4     @@qtde = 0
5
6     def initialize(marca, modelo, cor, tanque)
7         @marca = marca
8         @modelo = modelo
9         @cor = cor
10        @tanque = tanque

```

```
11     @@qtde += 1
12   end
13
14   def to_s
15     "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
16   end
17
18   def self.qtde
19     @@qtde
20   end
21 end
22
23 class NovoCarro < Carro
24   def to_s
25     "Marca nova:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
26   end
27 end
28
29 carro1 = Carro.new(:chevrolet, :corsa, :preto, 50)
30 carro2 = Carro.new(:chevrolet, :corsa, :prata, 50)
31 novo_carro = NovoCarro.new(:volks, :gol, :azul, 42)
32
33 puts carro1
34 puts carro2
35 puts novo_carro
36 puts Carro.qtde
37 puts NovoCarro.qtde
38
39 $ ruby code/carro11.rb
40 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
41 Marca:chevrolet Modelo:corsa Cor:prata Tanque:50
42 Marca nova:volks Modelo:gol Cor:azul Tanque:42
43 3
44 3
```

Poderíamos ter modificado para usar o método super:

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque
3   attr_accessor :cor
4   @@qtde = 0
5
6   def initialize(marca,modelo,cor,tanque)
7     @marca = marca
8     @modelo = modelo
9     @cor = cor
10    @tanque = tanque
11    @@qtde += 1
12  end
13
14  def to_s
15    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
16  end
17 end
18
19 class NovoCarro < Carro
20   def to_s
21     "Novo Carro: "+super
22   end
23 end
24
25 carro = Carro.new(:chevrolet,:corsa,:preto,50)
26 novo_carro = NovoCarro.new(:volks,:gol,:azul,42)
27
28 puts carro
29 puts novo_carro
30
31 $ ruby code/carro12.rb
32 Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
33 Novo Carro: Marca:volks Modelo:gol Cor:azul Tanque:42
```

O método `super` chama o mesmo método da classe pai, e tem dois comportamentos:

1. Sem parênteses, ele envia os mesmos argumentos recebidos pelo método corrente para o método pai.
2. Com parênteses, ele envia os argumentos selecionados.

Podemos ver como enviar só os selecionados:

```
1 class Teste
2   def metodo(parametro1)
3     puts parametro1
4   end
5 end
6
7 class NovoTeste < Teste
8   def metodo(parametro1,parametro2)
9     super(parametro1)
10    puts parametro2
11  end
12 end
13
14 t1 = Teste.new
15 t2 = NovoTeste.new
16 t1.metodo(1)
17 t2.metodo(2,3)
18
19 $ ruby code/supermeth.rb
20 1
21 2
22 3
```



Dica

Podemos utilizar um *hook* (“gancho”) para descobrir quando uma classe herda de outra:

```
1 class Pai
2   def self.inherited(child)
3     puts "#{child} herdando de #{self}"
4   end
5 end
6
7 class Filha < Pai
8   end
9
10 $ ruby code/inherited.rb
11 Filha herdando de Pai
```



Dica

Se estivermos com pressa e não quisermos fazer uma declaração completa de uma classe com seus *readers*, *writers* ou *accessors*, podemos herdar de uma Struct (lembram dela?) com alguns atributos da seguinte maneira:

```
1 class Carro < Struct.new(:marca, :modelo, :cor, :tanque)
2   def to_s
3     "Marca: #{marca} modelo: #{modelo} cor: #{cor} tanque: #{tanque}"
4   end
5 end
6 fox = Carro.new(:vw, :fox, :verde, 45)
7 puts fox
8 => Marca: vw modelo: fox cor: verde tanque: 45
```

Duplicando de modo raso e profundo

Sabemos que os valores são transferidos por referência, e se quisermos criar novos objetos baseados em alguns existentes? Para esses casos, podemos duplicar um objeto usando `dup`, gerando um novo objeto:

```
1 c1 = Carro.new
2 => #<Carro:0x9f0e138>
3 c2 = c1
4 => #<Carro:0x9f0e138>
5 c3 = c1.dup
6 => #<Carro:0x9f1d41c>
7 c1.object_id
8 => 83390620
9 c2.object_id
10 => 83390620
11 c3.object_id
12 => 83421710
```

Essa funcionalidade está implementada automaticamente para os objetos que são instâncias da nossa classe, mas fica uma dica: existem casos em que precisamos ter propriedades diferentes ao efetuar a cópia, como por exemplo, a variável de instância `@criado`, onde se utilizarmos `dup`, vai ser duplicada e não vai refletir a data e hora que esse novo objeto foi criado através da duplicação do primeiro:

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque, :criado
3   attr_accessor :cor
4
5   def initialize(marca,modelo,cor,tanque)
6     @marca = marca
7     @modelo = modelo
8     @cor = cor
9     @tanque = tanque
10    @criado = Time.now
11  end
12
13  def to_s
14    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
15  end
16 end
17
18 carro = Carro.new(:chevrolet,:corsa,:preto,50)
19 puts carro.criado
20 sleep 1
21
22 outro_carro = carro.dup
23 puts outro_carro.criado
24
25 $ruby dup.rb
26 2011-06-29 22:36:10 -0300
27 2011-06-29 22:36:10 -0300
```

Apesar de esperarmos 1 segundo utilizando o método `sleep`, o valor de `@criado` na cópia do objeto feita com `dup` permaneceu o mesmo. Para evitar isso, utilizamos `initialize_copy` na nossa classe, que vai ser chamado quando o objeto for duplicado, atualizando o valor da variável de instância `@criado_em`:

```
1 class Carro
2   attr_reader :marca, :modelo, :tanque, :criado
3   attr_accessor :cor
4
5   def initialize(marca,modelo,cor,tanque)
6     @marca = marca
7     @modelo = modelo
8     @cor = cor
9     @tanque = tanque
10    @criado = Time.now
11  end
12
13  def initialize_copy(original)
```

```
14     puts "criado objeto novo #{self.object_id} duplicado de #{original.ob\
15 ject_id}"
16     @criado = Time.now
17   end
18
19   def to_s
20     "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
21   end
22 end
23
24 carro = Carro.new(:chevrolet, :corsa, :preto, 50)
25 puts carro.criado
26 puts carro.object_id
27 sleep 1
28
29 outro_carro = carro.dup
30 puts outro_carro.criado
31 puts outro_carro.object_id
32
33 $ ruby code/initializcopy.rb
34 2011-06-29 22:36:10 -0300
35 83042330
36 criado objeto novo 82411250 duplicado de 83042330
37 2011-06-29 22:36:11 -0300
38 82411250
```

Agora a data e hora de criação/duplicação do objeto ficaram corretas.

Vale lembrar que cópias de objetos em Ruby usando dup são feitas usando o conceito de **shallow copy**, que duplica um objeto mas não os objetos referenciados dentro dele. Vamos ver um exemplo:

```
1 class A
2   attr_reader :outro
3
4   def initialize(outro=nil)
5     @outro = outro
6   end
7
8   def show
9     puts "Estou em #{self.class.name}, #{object_id}"
10    puts "Outro: #{@outro.object_id}" if !@outro.nil?
11  end
12 end
13
14 class B < A
```

```
15 end
16
17 a = A.new
18 b = B.new(a)
19
20 a.show
21 b.show
22
23 b2 = b.dup
24 b2.show
```

Rodando o programa:

```
1 $ ruby code/shallow.rb
2 Estou em A, 75626430
3 Estou em B, 75626420
4 Outro: 75626430 <===== aqui!
5 Estou em B, 75626300
6 Outro: 75626430 <===== aqui!
```

Pudemos ver que o objeto que consta na variável `b` foi duplicado, porém o objeto que consta na referência em `a` continua o mesmo em `b2`!

Para evitar esse tipo de coisa, precisamos do conceito de **deep copy**, que irá duplicar o objeto e os objetos dentro dele, retornando objetos totalmente novos.

Em Ruby isso pode ser alcançado através de **serialização** utilizando `Marshal`, armazenando os objetos como um fluxo de dados binários e depois restaurando todos em posições de memória totalmente novas:

```
1 class A
2   attr_accessor :outro
3
4   def initialize(outro=nil)
5     @outro = outro
6   end
7
8   def show
9     puts "Estou em #{self.class.name}, #{object_id}"
10    puts "Outro: #{@outro.object_id}" if !@outro.nil?
11  end
12 end
13
14 class B < A
15 end
16
```

```
17 a = A.new
18 b = B.new(a)
19
20 a.show
21 b.show
22
23 b2 = Marshal.load(Marshal.dump(b))
24 b2.show
```

Rodando o programa:

```
1 $ ruby code/deep.rb
2 Estou em A, 74010500
3 Estou em B, 74010490
4 Outro: 74010500 <===== aqui!
5 Estou em B, 74010330
6 Outro: 74010300 <===== aqui!
```

Brincando com métodos dinâmicos e hooks

Podemos emular o comportamento de uma `OpenStruct` utilizando o método `method_missing`, que é chamado caso o seu objeto o tenha declarado, sempre que ocorrer uma exceção do tipo `NoMethodError`, ou seja, quando o método que tentamos acessar não existe:

```
1 class Teste
2   def method_missing(meth,value=nil) # *args,&block
3     sanitized = meth.to_s.split("=").first
4     if meth =~ /=$/
5       self.class.send(:define_method, meth)
6       { |val| instance_variable_set("@#{sanitized}", val) }
7       self.send(meth, value)
8     else
9       self.class.send(:define_method, sanitized)
10      { instance_variable_get("@#{sanitized}") }
11      self.send(meth)
12    end
13  end
14 end
15
16 t = Teste.new
17 t.oi = "oi, mundo!"
18 puts t.oi
19
20 puts t.hello
```

```

21 t.hello = "hello, world!"
22 puts t.hello
23
24 $ ruby code/methmissing.rb
25 oi, mundo!
26 hello, world!

```

Vamos aproveitar e testar dois *hooks* para métodos, `method_added` e `method_removed`:

```

1 # encoding: utf-8
2 class Teste
3   def self.method_added(meth)
4     puts "Adicionado o método #{meth}"
5   end
6
7   def self.method_removed(meth)
8     puts "Removido o método #{meth}"
9   end
10 end
11
12 t = Teste.new
13 t.class.send(:define_method, "teste") { puts "teste!" }
14 t.teste
15 t.class.send(:remove_method, :teste)
16 t.teste
17
18 $ ruby code/hooksmeth.rb
19 Adicionado o método teste
20 teste!
21 Removido o método teste
22 code/hooksmeth.rb:16:in '<main>': undefined method 'teste' for #<Teste:0x9f3d\
23 12c> (NoMethodError)

```

Podemos definir “métodos fantasmas” (*ghost methods*, buuuuu!), brincando com `method_missing`:

```

1 # encoding: utf-8
2 class Teste
3   def method_missing(meth)
4     puts "Não sei o que fazer com a sua requisição: #{meth}"
5   end
6 end
7 t = Teste.new
8 t.teste
9
10 $ ruby code/ghost.rb
11 Não sei o que fazer com a sua requisição: teste

```

Manipulando métodos que se parecem com operadores

Vamos imaginar que temos uma classe chamada `CaixaDeParafusos` e queremos algum jeito de fazer ela interagir com outra, por exemplo, adicionando o conteúdo de um outra (e esvaziando a que ficou sem conteúdo). Podemos fazer coisas do tipo:

```
1 class CaixaDeParafusos
2   attr_reader :quantidade
3
4   def initialize(quantidade)
5     @quantidade = quantidade
6   end
7
8   def to_s
9     "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
10  end
11
12  def +(outra)
13    CaixaDeParafusos.new(@quantidade+outra.quantidade)
14  end
15 end
16
17 caixa1 = CaixaDeParafusos.new(10)
18 caixa2 = CaixaDeParafusos.new(20)
19 caixa3 = caixa1 + caixa2
20
21 puts caixa1
22 puts caixa2
23 puts caixa3
24
25 $ ruby code/caixa1.rb
26 Quantidade de parafusos na caixa 69826490: 10
27 Quantidade de parafusos na caixa 69826480: 20
28 Quantidade de parafusos na caixa 69826470: 30
```

Mas espera aí! Se eu somei uma caixa com a outra em uma terceira, não deveria ter sobrado nada nas caixas originais, mas ao invés disso elas continuam intactas. Precisamos zerar a quantidade de parafusos das outras caixas:

```
1 class CaixaDeParafusos
2   attr_reader :quantidade
3
4   def initialize(quantidade)
5     @quantidade = quantidade
6   end
7
8   def to_s
9     "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
10  end
11
12  def +(outra)
13    CaixaDeParafusos.new(@quantidade+outra.quantidade)
14    @quantidade = 0
15    outra.quantidade = 0
16  end
17 end
18
19 caixa1 = CaixaDeParafusos.new(10)
20 caixa2 = CaixaDeParafusos.new(20)
21 caixa3 = caixa1 + caixa2
22
23 puts caixa1
24 puts caixa2
25 puts caixa3
26
27 $ ruby code/caixa2.rb
28 code/caixa2.rb:15:in '+': undefined method 'quantidade=' for Quantidade de pa\
29 rafusos na caixa 74772290: 20:CaixaDeParafusos (NoMethodError)
30 from code/caixa2.rb:21:in '<main>'
```

Parece que ocorreu um erro ali, mas está fácil de descobrir o que é. Tentamos acessar a variável de instância da **outra caixa** enviada como parâmetro mas não temos um `attr_writer` para ela!

Mas espera aí: só queremos que essa propriedade seja alterada quando efetuando alguma operação com outra caixa de parafusos ou alguma classe filha, e não seja acessada por qualquer outra classe. Nesse caso, podemos usar um **método protegido**:

```
1 class CaixaDeParafusos
2   protected
3   attr_writer :quantidade
4
5   public
6   attr_reader :quantidade
7
8   def initialize(quantidade)
9     @quantidade = quantidade
10  end
11
12  def to_s
13    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
14  end
15
16  def +(outra)
17    nova = CaixaDeParafusos.new(@quantidade+outra.quantidade)
18    @quantidade = 0
19    outra.quantidade = 0
20    nova
21  end
22 end
23
24 caixa1 = CaixaDeParafusos.new(10)
25 caixa2 = CaixaDeParafusos.new(20)
26 caixa3 = caixa1 + caixa2
27
28 puts caixa1
29 puts caixa2
30 puts caixa3
31
32 $ ruby code/caixa3.rb
33
34 Quantidade de parafusos na caixa 81467020: 0
35 Quantidade de parafusos na caixa 81467010: 0
36 Quantidade de parafusos na caixa 81467000: 30
```

Agora pudemos ver que tudo funcionou perfeitamente, pois utilizamos `protected` antes de inserir o `attr_writer`. Os **modificadores de controle de acesso de métodos** são:

1. **Públicos (public)** - Podem ser acessados por qualquer método em qualquer objeto.
2. **Privados (private)** - Só podem ser chamados dentro de seu próprio objeto, mas nunca é possível acessar um método privado de outro objeto, mesmo se o objeto que chama seja uma sub-classe de onde o método foi definido.
3. **Protegidos (protected)** - Podem ser acessados em seus descendentes.



Dica

Usando a seguinte analogia para lembrar do acesso dos métodos: vamos supor que você seja dono de um restaurante. Como você não quer que seus fregueses fiquem apertados você manda fazer um banheiro para o pessoal, mas nada impede também que apareça algum maluco da rua apertado, entre no restaurante e use seu banheiro (ainda mais se ele tiver 2 metros de altura, 150 kg e for lutador de alguma arte marcial). Esse banheiro é **público**. Para seus empregados, você faz um banheirinho mais caprichado, que só eles tem acesso. Esse banheiro é **protegido**, sendo que só quem é do restaurante tem acesso. Mas você sabe que tem um empregado seu lá que tem uns problemas e ao invés de utilizar o banheiro, ele o **inutiliza**. Como você tem enjoo com esse tipo de coisa, manda fazer um banheiro **privado** para você, que só você pode usar.

Agora vamos supor que queremos dividir uma caixa em caixas menores com conteúdos fixos e talvez o resto que sobrar em outra. Podemos usar o método /:

```

1 class CaixaDeParafusos
2   protected
3   attr_writer :quantidade
4
5   public
6   attr_reader :quantidade
7
8   def initialize(quantidade)
9     @quantidade = quantidade
10  end
11
12  def to_s
13    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
14  end
15
16  def +(outra)
17    nova = CaixaDeParafusos.new(@quantidade+outra.quantidade)
18    @quantidade = 0
19    outra.quantidade = 0
20    nova
21  end
22
23  def /(quantidade)
24    caixas = Array.new(@quantidade/quantidade,quantidade)
25    caixas << @quantidade%quantidade if @quantidade%quantidade>0
26    @quantidade = 0
27    caixas.map {|quantidade| CaixaDeParafusos.new(quantidade)}
28  end
29 end
30
31 caixa1 = CaixaDeParafusos.new(10)

```

```
32 caixa2 = CaixaDeParafusos.new(20)
33 caixa3 = caixa1 + caixa2
34
35 puts caixa3 / 8
36
37 $ ruby code/caixa4.rb
38 Quantidade de parafusos na caixa 67441310: 8
39 Quantidade de parafusos na caixa 67441300: 8
40 Quantidade de parafusos na caixa 67441290: 8
41 Quantidade de parafusos na caixa 67441280: 6
```

Ou podemos simplesmente pedir para dividir o conteúdo em X caixas menores, distribuindo uniformemente o seu conteúdo:

```
1 class CaixaDeParafusos
2   protected
3   attr_writer :quantidade
4
5   public
6   attr_reader :quantidade
7
8   def initialize(quantidade)
9     @quantidade = quantidade
10  end
11
12  def to_s
13    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
14  end
15
16  def +(outra)
17    nova = CaixaDeParafusos.new(@quantidade+outra.quantidade)
18    @quantidade = 0
19    outra.quantidade = 0
20    nova
21  end
22
23  def /(quantidade)
24    caixas = Array.new(quantidade,@quantidade/quantidade)
25    (@quantidade%quantidade).times {|indice| caixas[indice] += 1}
26    @quantidade = 0
27    caixas.map {|quantidade| CaixaDeParafusos.new(quantidade)}
28  end
29 end
30
31 caixa1 = CaixaDeParafusos.new(10)
32 caixa2 = CaixaDeParafusos.new(20)
```

```
33 caixa3 = caixa1 + caixa2
34
35 puts caixa3 / 4
36
37 $ ruby code/caixa5.rb
38 Quantidade de parafusos na caixa 81385900: 8
39 Quantidade de parafusos na caixa 81385890: 8
40 Quantidade de parafusos na caixa 81385880: 7
41 Quantidade de parafusos na caixa 81385870: 7
```

Closures

Vamos fazer um gancho aqui falando em classes e métodos para falar um pouco de **closures**. Closures são funções anônimas com escopo fechado que mantém o estado do ambiente em que foram criadas.

Os blocos de código que vimos até agora eram todos closures, mas para dar uma dimensão do fato de closures guardarem o seu ambiente podemos ver:

```
1 def cria_contador(inicial, incremento)
2   contador = inicial
3   lambda { contador += incremento }
4 end
5
6 meu_contador = cria_contador(0,1)
7
8 puts meu_contador.call
9 puts meu_contador.call
10 puts meu_contador.call
11
12 $ ruby code/closures.rb
13 1
14 2
15 3
```

A Proc foi criada pela lambda na linha 3, que guardou a referência para a variável contador mesmo depois que saiu do escopo do método cria_contador.

Módulos

Mixins

Ruby tem herança única, como vimos quando criamos nossas próprias classes, mas conta com o conceito de módulos (também chamados nesse caso de mixins) para a incorporação de funcionalidades adicionais. Para utilizar um módulo, utilizamos `include`:

```
1 class Primata
2   def come
3     puts "Nham!"
4   end
5
6   def dorme
7     puts "Zzzzzz..."
8   end
9 end
10
11 class Humano < Primata
12   def conecta_na_web
13     puts "Login ... senha ..."
14   end
15 end
16
17 module Ave
18   def voa
19     puts "Para o alto, e avante!"
20   end
21 end
22
23 class Mutante < Humano
24   include Ave
25 end
26
27 mutante = Mutante.new
28 mutante.come
29 mutante.dorme
30 mutante.conecta_na_web
31 mutante.voa
32
33 $ ruby code/mod1.rb
34 Nham!
```

```
35 Zzzzzz...
36 Login ... senha ...
37 Para o alto, e avante!
```

Como podemos ver, podemos misturar várias características de um módulo em uma classe. Isso poderia ter sido feito para apenas uma instância de um objeto usando `extend`, dessa forma:

```
1 class Primata
2   def come
3     puts "Nham!"
4   end
5
6   def dorme
7     puts "Zzzzzz..."
8   end
9 end
10
11 class Humano < Primata
12   def conecta_na_web
13     puts "Login ... senha ..."
14   end
15 end
16
17 module Ave
18   def voa
19     puts "Para o alto, e avante!"
20   end
21 end
22
23 class Mutante < Humano
24 end
25
26 mutante = Mutante.new
27 mutante.extend(Ave)
28 mutante.come
29 mutante.dorme
30 mutante.conecta_na_web
31 mutante.voa
32
33 mutante2 = Mutante.new
34 mutante2.voa
35
36 $ ruby code/mod2.rb
37 Nham!
38 Zzzzzz...
39 Login ... senha ...
```

```

40 Para o alto, e avante!
41 code/mod2.rb:33:in '<main>': undefined method 'voa' for #<Mutante:0x855465c> \
42 (NoMethodError)

```



Dica

O método `extend` inclui os métodos de um módulo na *eingenclass* (classe fantasma, *singleton*, etc.) do objeto onde está sendo executado.

Uma coisa bem importante a ser notada é que quanto usamos `include` os métodos provenientes do módulo são incluídos nas **instâncias das classes**, e não nas **classes** em si. Se quisermos definir métodos de classes dentro dos módulos, podemos utilizar um outro *hook* chamado `included`, usando um módulo interno (???):

```

1 # encoding: utf-8
2 module TesteMod
3   module ClassMethods
4     def class_method
5       puts "Esse é um método da classe!"
6     end
7   end
8
9   def self.included(where)
10    where.extend(ClassMethods)
11  end
12
13  def instance_method
14    puts "Esse é um método de instância!"
15  end
16 end
17
18 class TesteCls
19   include TesteMod
20 end
21
22 t = TesteCls.new
23 t.instance_method
24 TesteCls.class_method
25
26 $ ruby code/mod7.rb
27 Esse é um método de instância!
28 Esse é um método da classe!

```

Os métodos dos módulos são inseridos nas procura dos métodos (*method lookup*) logo **antes** da classe que os incluiu.

Se incluirmos o módulo em uma classe, os métodos do módulo se tornam métodos das instâncias da classe. Se incluirmos o módulo na *eigenclass* da classe, se tornam métodos da classe. Se incluirmos em uma instância da classe, se tornam métodos *singleton* do objeto em questão.

Temos alguns comportamentos bem úteis usando mixins. Alguns nos pedem apenas um método para dar em troca vários outros. Se eu quisesse implementar a funcionalidade do módulo Comparable no meu objeto, eu só teria que fornecer um método `<=>` (*starship*, “navinha”) e incluir o módulo:

```
1 class CaixaDeParafusos
2   include Comparable
3   attr_reader :quantidade
4
5   def initialize(quantidade)
6     @quantidade = quantidade
7   end
8
9   def <=>(outra)
10    self.quantidade <=> outra.quantidade
11  end
12 end
13
14 caixa1 = CaixaDeParafusos.new(10)
15 caixa2 = CaixaDeParafusos.new(20)
16 caixa3 = CaixaDeParafusos.new(10)
17
18 puts caixa1 < caixa2
19 puts caixa2 > caixa3
20 puts caixa1 == caixa3
21 puts caixa3 > caixa2
22 puts caixa1.between?(caixa3,caixa2)
23
24 $ ruby code/mod3.rb
25 true
26 true
27 true
28 false
29 true
```

Com isso ganhamos os métodos `<`, `<=`, `==`, `>`, `>=` e `between?`. Vamos criar um iterador mixando o módulo Enumerable:

```
1  # encoding: utf-8
2  class Parafuso
3      attr_reader :polegadas
4
5      def initialize(polegadas)
6          @polegadas = polegadas
7      end
8
9      def <=>(outro)
10         self.polegadas <=> outro.polegadas
11     end
12
13     def to_s
14         "Parafuso #{object_id} com #{@polegadas}\n"
15     end
16 end
17
18 class CaixaDeParafusos
19     include Enumerable
20
21     def initialize
22         @parafusos = []
23     end
24
25     def <<(parafuso)
26         @parafusos << parafuso
27     end
28
29     def each
30         @parafusos.each {|numero| yield(numero) }
31     end
32 end
33
34 caixa = CaixaDeParafusos.new
35 caixa << Parafuso.new(1)
36 caixa << Parafuso.new(2)
37 caixa << Parafuso.new(3)
38
39 puts "o menor parafuso na caixa é: #{caixa.min}"
40 puts "o maior parafuso na caixa é: #{caixa.max}"
41 puts "os parafusos com medidas par são: #{caixa.select {|parafuso| parafuso.polegadas%2==0}.join(', ')}"
42 puts "duplicando a caixa: #{caixa.map {|parafuso| Parafuso.new(parafuso.polegadas*2)}}"
43
44 $ ruby code/mod4.rb
```

```
47 o menor parafuso na caixa é: Parafuso 72203410 com 1"
48 o maior parafuso na caixa é: Parafuso 72203390 com 3"
49 os parafusos com medidas par são: Parafuso 72203400 com 2"
50 duplicando a caixa: [Parafuso 72203110 com 2", Parafuso 72203100 com 4", Para\
51 fuso 72203090 com 6"]
```

Podemos ver como são resolvidas as chamadas de métodos utilizando `ancestors`:

```
1 class C
2   def x; "x"; end
3 end
4
5 module M
6   def x; '[' + super + ']'; end
7   def y; "y"; end
8 end
9
10 class C
11   include M
12 end
13
14 p C.ancestors # [C, M, Object, Kernel, BasicObject]
15 c = C.new
16 puts c.x
17 => x
18 puts c.y
19 => y
```

Reparem que o módulo foi inserido na cadeia de chamadas *após* a classe corrente, tanto que quando temos na classe um método com o mesmo nome que o do módulo, é chamado o método da classe.



Novidade em Ruby 2.0

A partir da versão 2, temos o método `prepend`, que insere o módulo *antes* na cadeia de chamada de métodos:

```
1 class C
2   def x; "x"; end
3 end
4
5 module M
6   def x; '[' + super + ']'; end
7   def y; "y"; end
8 end
9
10 class C
11   prepend M
12 end
13
14 p C.ancestors # => [M, C, Object, Kernel, BasicObject]
15 c = C.new
16 puts c.x # => [x]
17 puts c.y # => y
```

Outro ponto bem importante para se notar é que, se houverem métodos em comum entre os módulos inseridos, o **método do último módulo incluído é que vai valer**. Vamos fazer um arquivo chamado `overmod.rb` com o seguinte código:

```
1 # encoding: utf-8
2
3 module Automovel
4   def ligar
5     puts "ligando automóvel"
6   end
7 end
8
9 module Radio
10  def ligar
11    puts "ligando rádio"
12  end
13 end
14
15 class Carro
16   include Automovel
17   include Radio
18 end
19
```

```
20 c = Carro.new
21 c.ligar
```

Rodando o código:

```
1 $ ruby overmod.rb
2 ligando rádio
```

Pudemos ver que o módulo `Radio` foi incluído por último, conseqüentemente o seu método `ligar` é que foi utilizado. Isso é fácil de constatar verificando os ancestrais de `Carro`:

```
1 $ Carro.ancestors
2 => [Carro, Radio, Automovel, Object, Kernel, BasicObject]
```

Para chamar o método de `Automovel`, podemos explicitamente chamar o método dessa maneira, que faz um `bind` do método com o objeto corrente:

```
1 class Carro
2   include Automovel
3   include Radio
4
5   def ligar
6     Automovel.instance_method(:ligar).bind(self).call
7   end
8 end
```

Namespaces

Módulos também podem ser utilizados como **namespaces**, que nos permitem delimitar escopos e permitir a separação e resolução de identificadores, como classes e métodos, que sejam homônimos. Vamos pegar como exemplo um método chamado `comida_preferida`, que pode estar definido em várias classes **de mesmo nome**, porém em **módulos diferentes**:

```
1 module Paulista
2   class Pessoa
3     def comida_preferida
4       "pizza"
5     end
6   end
7 end
8
9 module Gaucho
10  class Pessoa
```

```
11     def comida_preferida
12         "churrasco"
13     end
14 end
15 end
16
17 pessoa1 = Paulista::Pessoa.new
18 pessoa2 = Gaucho::Pessoa.new
19
20 puts pessoa1.comida_preferida
21 puts pessoa2.comida_preferida
22
23 $ ruby code/mod5.rb
24 pizza
25 churrasco
```

Apesar de ambas as classes chamarem Pessoa e terem métodos chamados comida_preferida, elas estão separadas através de cada módulo em que foram definidas. É uma boa idéia utilizar namespaces quando criarmos algo com nome, digamos, comum, que sabemos que outras pessoas podem criar com os mesmos nomes. Em Java, por exemplo, [existe a convenção que um namespace pode ser um domínio invertido](#)¹², utilizando a *keyword* package, como por exemplo:

```
1 package com.eustaquiorangel.paulista;
```

Dando uma olhada em como resolvemos isso em Java:

```
1 // com/eustaquiorangel/paulista/Pessoa.java
2 package com.eustaquiorangel.paulista;
3 public class Pessoa {
4     public static String comidaPreferida() {
5         return "pizza";
6     }
7 }
8
9 // com/eustaquiorangel/gaucho/Pessoa.java
10 package com.eustaquiorangel.gaucho;
11 public class Pessoa {
12     public static String comidaPreferida() {
13         return "churrasco";
14     }
15 }
16
17 // Namespace.java
18 public class Namespace {
```

¹²<http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>

```
19     public static void main(String args[]) {
20         System.out.println(com.eustaquiorangel.paulista.Pessoa.comidaPreferid\
21 a());
22         System.out.println(com.eustaquiorangel.gaucho.Pessoa.comidaPreferida(\
23 ));
24     }
25 }
```

Está certo que cada arquivo tem que ser criado na estrutura de diretórios de acordo com o nome do package e outros detalhes, mas, depois de compilados (e opcionalmente empacotados), funciona direitinho:

```
1 $ javac -cp ..//* Namespace.java
2 $ java -cp ..//* Namespace
3 pizza
4 churrasco
```

Podemos implementar algumas funcionalidades interessantes com módulos, por exemplo, criar uma classe Singleton¹³:

```
1 # encoding: utf-8
2 require "singleton"
3
4 class Teste
5     include Singleton
6 end
7
8 begin
9     Teste.new
10 rescue => exception
11     puts "Não consegui criar usando new: #{exception}"
12 end
13
14 puts Teste.instance.object_id
15 puts Teste.instance.object_id
16
17 $ ruby code/mod6.rb
18 Não consegui criar usando new: private method 'new' called for Teste:Class
19 69705530
20 69705530
```

¹³<http://pt.wikipedia.org/wiki/Singleton>

TracePoint



Atenção!

A classe `TracePoint` só está disponível a partir da versão 2.0 de Ruby. Em outras versões, comportamento similar pode ser obtido através do método `set_trace_func`¹⁴.

A classe `TracePoint` nos permite coletar informações durante a execução do nosso programa, interceptando vários tipos (ou todos) de eventos que ocorrem. Os eventos são:

- `:line` - executar código em uma nova linha
- `:class` - início da definição de uma classe ou módulo
- `:end` - fim da definição de uma classe ou módulo
- `:call` - chamada de um método Ruby
- `:return` - retorno de um método Ruby
- `:c_call` - chamada de uma rotina em C
- `:c_return` - retorno de uma rotina em C
- `:raise` - exceção disparada
- `:b_call` - início de um bloco
- `:b_return` - fim de um bloco
- `:thread_begin` - início de uma `Thread`
- `:thread_end` - fim de uma `Thread`

Quando interceptamos alguns desses eventos, temos na `TracePoint` as seguintes informações disponíveis:

- `binding` - o *binding* corrente do evento
- `defined_class` - a classe ou módulo do método chamado
- `event` - tipo do evento
- `inspect` - uma `String` com o *status* de forma legível
- `lineno` - o número da linha do evento
- `method_id` - o nome do método sendo chamado
- `path` - caminho do arquivo sendo executado
- `raised_exception` - exceção que foi disparada
- `return_value` - valor de retorno
- `self` - o objeto utilizado durante o evento

Para ativarmos a `TracePoint`, criamos uma nova instância da classe, com os eventos que queremos monitorar, e logo após chamamos o método `enable`. Vamos ver como funciona no arquivo `tpoint.rb`:

¹⁴http://ruby-doc.org/core-2.0/Kernel.html#method-i-set_trace_func

```

1 TracePoint.new(:class, :end, :call) do |tp|
2   puts "Disparado por #{tp.self} no arquivo #{tp.path} na linha #{tp.lineno}\
3   }}"
4 end.enable
5
6 module Paulista
7   class Pessoa
8     end
9 end
10 p = Paulista::Pessoa.new

```

Rodando o programa:

```

1 $ ruby tpoint.rb
2 Disparado por Paulista no arquivo tpoint.rb na linha 5
3 Disparado por Paulista::Pessoa no arquivo tpoint.rb na linha 6
4 Disparado por Paulista::Pessoa no arquivo tpoint.rb na linha 7
5 Disparado por Paulista no arquivo tpoint.rb na linha 8

```

A classe `TracePoint` nos permite fazer algumas coisas bem legais no nosso código. Como exemplo disso, vi em um [Metacast](#)¹⁵ um exemplo para tentar definir uma interface¹⁶ em Ruby, e dei uma mexida nele [para ficar assim](#)¹⁷:

```

1 module AbstractInterface
2   class NotImplementedError < StandardError
3     def initialize(*methods)
4       super "You must implement the following methods: #{methods.join(', '\
5     }}"
6   end
7 end
8
9 def AbstractInterface.check_methods(klass, other, methods)
10  return if other.class==Module
11  TracePoint.new(:end) do |tp|
12    return if tp.self!=other || methods.nil?
13    missing = methods.select {|method| !other.instance_methods.include?(\
14  method)}
15    raise NotImplementedError.new(missing) if missing.any?
16  end.enable
17 end
18
19 module ClassMethods

```

¹⁵<http://www.metacasts.tv/casts/tracepoint>

¹⁶[http://pt.wikipedia.org/wiki/Interface_\(programa%C3%A7%C3%A3o\)](http://pt.wikipedia.org/wiki/Interface_(programa%C3%A7%C3%A3o))

¹⁷<https://gist.github.com/taq/5863818>

```
20     def abstract_method(*args)
21         return @abstract_method if !args
22         @abstract_method ||= []
23         @abstract_method.push(*args)
24     end
25     def included(other)
26         AbstractInterface.check_methods(self, other, @abstract_method)
27     end
28     def check_methods(klass, other, methods)
29         AbstractInterface.check_methods(klass, other, methods)
30     end
31 end
32
33 def self.included(other)
34     check_methods(self, other, @abstract_method)
35     other.extend ClassMethods
36 end
37 end
38
39 module FooBarInterface
40     include AbstractInterface
41     abstract_method :foo, :bar
42 end
43
44 module BazInterface
45     include AbstractInterface
46     abstract_method :baz
47 end
48
49 class Test
50     include FooBarInterface
51     include BazInterface
52     def foo
53         puts "foo"
54     end
55     def bar
56         puts "bar"
57     end
58     def baz
59         puts "baz"
60     end
61 end
62 t = Test.new
63 t.foo
64 t.bar
65 t.baz
```

Tentem comentar alguns dos métodos definidos em `Test` e rodar o programa, vai ser disparada uma exceção do tipo `NotImplementedError`!

Antes de ver mais uma funcionalidade bem legal relacionada à módulos, vamos ver como fazemos para instalar pacotes novos que vão nos prover essas funcionalidades, através das **RubyGems**.

Instalando pacotes novos através do RubyGems

O **RubyGems** é um projeto feito para gerenciar as **gems**, que são pacotes com aplicações ou bibliotecas Ruby, com nome e número de versão. O suporte à gems já se encontra instalado, pois instalamos o nosso interpretador Ruby com a RVM.

Se não estivermos utilizando a RVM, apesar de alguns sistemas operacionais já terem pacotes prontos, recomenda-se instalar a partir do código-fonte. Para isso, é necessário ter um interpretador de Ruby instalado e seguir os seguintes passos (lembrando de verificar qual é a última versão disponível em <http://rubygems.org>¹⁸ e executar os comandos seguintes como *root* ou usando *sudo*):

```
1 wget http://production.cf.rubygems.org/rubygems/rubygems-1.8.5.tgz
2 tar xvzf rubygems-1.8.5.tgz
3 cd rubygems-1.8.5
4 ruby setup.rb
5 gem -v => 1.8.5
```



Dica

Certifique-se de ter instalado a biblioteca **zlib** (e, dependendo da sua distribuição, o pacote **zlib-devel** também).

Após instalado, vamos dar uma olhada em algumas opções que temos, sempre usando a opção como parâmetro do comando `gem`:

- **list** - Essa opção lista as gems atualmente instaladas. Por não termos ainda instalado nada, só vamos encontrar os sources do RubyGems.
- **install** - Instala a gem requisitada. No nosso caso, vamos instalar a **gem** `memoize`, que vamos utilizar logo a seguir:

```
1 gem install memoize
2 Successfully installed memoize-1.3.1
3 Installing ri documentation for memoize-1.3.1...
4 Installing RDoc documentation for memoize-1.3.1...
```

- **update** - Atualiza a *gem* específica ou todas instaladas. Você pode usar `--include-dependencies` para instalar todas as dependências necessárias.

¹⁸<http://rubygems.org>

- **outdated** - Lista as *gems* que precisam de atualização no seu computador.
- **cleanup** - Essa é uma opção muito importante após rodar o `update`. Para evitar que algo se quebre por causa do uso de uma versão específica de um *gem*, o RubyGems mantém todas as versões antigas até que você execute o comando `cleanup`. Mas preste atenção se alguma aplicação não precisa de uma versão específica - e antiga - de alguma *gem*.
- **uninstall** - Desinstala uma *gem*.
- **search** - Procura uma determinada palavra em uma *gem*:

```
1 gem search -l memo
2 *** LOCAL GEMS ***
3 memoize (1.2.3)
```

Podem ser especificadas chaves para procurar as *gems* locais (-l) e remotas (-r). Verifique qual o comportamento padrão da sua versão do Ruby executando `search` sem nenhuma dessas chaves.



Depois de instalado, para atualizar o próprio RubyGems use a opção:

```
1 gem update --system
```

Instalamos essa *gem* específica para verificar uma funcionalidade muito interessante, a *memoization*, que acelera a velocidade do programa armazenando os resultados de chamadas aos métodos para recuperação posterior.

Se estivermos utilizando uma versão de Ruby **anterior** a 1.9.x, antes de mais nada temos que indicar, no início do programa, que vamos usar as *gems* através de

```
1 require "rubygems"
```

Sem isso o programa não irá saber que desejamos usar as *gems*, então “no-no-no se esqueça disso, Babalu!”. Algumas instalações e versões de Ruby da 1.9.x já carregam as RubyGems automaticamente, mas não custa prevenir.



Dica

Não confunda `require` com `include` ou com o método `load`. Usamos `include` para inserir os módulos, `require` para carregar “bibliotecas” de código e `load` para carregar e executar código, que pode ter o código carregado como um módulo anônimo, que é imediatamente destruído após o seu uso, se enviarmos `true` como o segundo argumento. Vamos ver sem utilizar `true`:

```
1  $ cat load1.rb
2  # encoding: utf-8
3
4  class Teste
5      def initialize
6          puts "comportamento padrão"
7      end
8  end
9
10 load("load2.rb")
11 Teste.new
12
13 $ cat load2.rb
14 # encoding: utf-8
15
16 class Teste
17     def initialize
18         puts "comportamento reescrito"
19     end
20 end
21
22 $ ruby load1.rb
23 => comportamento reescrito
24
25 agora, se utilizarmos `load("load2.rb",true)`:
26
27 => comportamento padrão
```

Agora vamos dar uma olhada na tal da *memoization*. Vamos precisar de um método com muitas chamadas, então vamos usar um recursivo. Que tal a sequência de Fibonacci ¹⁹? Primeiro vamos ver sem usar *memoization*:

¹⁹http://en.wikipedia.org/wiki/Fibonacci_number

```
1 def fib(numero)
2   return numero if numero < 2
3   fib(numero-1)+fib(numero-2)
4 end
5
6 puts Time.now
7 puts fib(ARGV[0].to_i)
8 puts Time.now
9
10 $ ruby code/memo1.rb 10
11 2011-06-30 20:16:08 -0300
12 55
13 2011-06-30 20:16:08 -0300
14
15 $ ruby code/memo1.rb 20
16 2011-06-30 20:16:10 -0300
17 6765
18 2011-06-30 20:16:10 -0300
19
20 $ ruby code/memo1.rb 30
21 832040
22 2011-06-30 20:16:12 -0300
23
24 $ ruby code/memo1.rb 40
25 2011-06-30 20:16:13 -0300
26 102334155
27 2011-06-30 20:16:56 -0300
```

Recomendo não usar um número maior que 40 ali não se vocês quiserem dormir em cima do teclado antes de acabar de processar. ;-)

Vamos fazer uma experiência e fazer o mesmo programa em Java:

```
1 import java.text.SimpleDateFormat;
2 import java.util.Calendar;
3
4 public class Fib {
5   public static long calcula(int numero) {
6     if(numero<2)
7       return numero;
8     return calcula(numero-1)+calcula(numero-2);
9   }
10
11   public static void main(String args[]) {
12     SimpleDateFormat fmt = new SimpleDateFormat("dd/MM/yyyy H:mm:ss");
13     System.out.println(fmt.format(Calendar.getInstance().getTime()));
```

```
14     System.out.println(calcula(Integer.parseInt(args[0])));
15     System.out.println(fmt.format(Calendar.getInstance().getTime()));
16 }
17 }
18
19 $ java Fib 10
20 30/06/2011 20:18:26
21 55
22 30/06/2011 20:18:26
23
24 $ java Fib 20
25 30/06/2011 20:18:28
26 6765
27 30/06/2011 20:18:28
28
29 $ java Fib 30
30 30/06/2011 20:18:29
31 832040
32 30/06/2011 20:18:29
33
34 $ java Fib 40
35 30/06/2011 20:18:31
36 102334155
37 30/06/2011 20:18:32
```

Bem mais rápido hein? Mas agora vamos refazer o código em Ruby, usando *memoization*:

```
1 require "rubygems"
2 require "memoize"
3 include Memoize
4
5 def fib(numero)
6     return numero if numero < 2
7     fib(numero-1)+fib(numero-2)
8 end
9 memoize(:fib)
10
11 puts Time.now
12 puts fib(ARGV[0].to_i)
13 puts Time.now
14
15 $ ruby code/memo2.rb 40
16 2011-06-30 20:19:36 -0300
17 102334155
18 2011-06-30 20:19:36 -0300
19
```

```
20 $ ruby code/memo2.rb 50
21 2011-06-30 20:19:39 -0300
22 12586269025
23 2011-06-30 20:19:39 -0300
24
25 $ ruby code/memo2.rb 100
26 2011-06-30 20:19:41 -0300
27 354224848179261915075
28 2011-06-30 20:19:41 -0300
```

Uau! Se quiserem trocar aquele número de 40 para 350 agora pode, sério! :-)) E ainda dá para otimizar mais se indicarmos um arquivo (nesse caso, chamado memo.cache) para gravar os resultados:

```
1 require "rubygems"
2 require "memoize"
3 include Memoize
4
5 def fib(numero)
6   return numero if numero < 2
7   fib(numero-1)+fib(numero-2)
8 end
9 memoize(:fib, "memo.cache")
10
11 puts Time.now
12 puts fib(ARGV[0].to_i)
13 puts Time.now
14
15 $ ruby code/memo3.rb 100
16 2011-06-30 20:21:22 -0300
17 354224848179261915075
18 2011-06-30 20:21:22 -0300
19
20 $ ruby code/memo3.rb 200
21 2011-06-30 20:21:25 -0300
22 280571172992510140037611932413038677189525
23 2011-06-30 20:21:25 -0300
24
25 $ ruby code/memo3.rb 350
26 2011-06-30 20:21:28 -0300
27 6254449428820551641549772190170184190608177514674331726439961915653414425
28 2011-06-30 20:21:28 -0300
```

**Dica**

Podemos fazer o mesmo comportamento de memoization utilizando uma Hash da seguinte maneira:

```

1  fib = Hash.new{ |h, n| n < 2 ? h[n] = n : h[n] = h[n - 1] + h[n - 2]}
2
3  puts Time.now; puts fib[10]; puts Time.now
4  puts Time.now; puts fib[100]; puts Time.now
5  2011-11-24 18:12:55 -0200
6  55
7  2011-11-24 18:12:55 -0200
8  2011-11-24 18:12:59 -0200
9  354224848179261915075
10 2011-11-24 18:12:59 -0200

```

**Outra dica**

Podemos calcular uma aproximação de um número de Fibonacci usando a seguinte equação, onde n é a sequência que queremos descobrir e Φ (*Phi*) é a “proporção áurea”:

$$\Phi^n / \sqrt{5}$$

Podemos definir o cálculo da seguinte forma:

```

1  phi = (Math.sqrt(5)/2)+0.5      => 1.618033988749895
2  ((phi**1)/Math.sqrt(5)).round  => 1
3  ((phi**2)/Math.sqrt(5)).round  => 1
4  ((phi**3)/Math.sqrt(5)).round  => 2
5  ((phi**4)/Math.sqrt(5)).round  => 3
6  ((phi**5)/Math.sqrt(5)).round  => 5
7  ((phi**6)/Math.sqrt(5)).round  => 8
8  ((phi**7)/Math.sqrt(5)).round  => 13
9  ((phi**8)/Math.sqrt(5)).round  => 21
10 ((phi**9)/Math.sqrt(5)).round  => 34
11 ((phi**10)/Math.sqrt(5)).round => 55
12 ((phi**40)/Math.sqrt(5)).round => 102334155
13 ((phi**50)/Math.sqrt(5)).round => 12586269025
14 ((phi**100)/Math.sqrt(5)).round => 354224848179263111168

```

Podemos ver que, quanto maior o número, mais ocorre algum pequeno desvio.

Threads

Uma linguagem de programação que se preze tem que ter suporte à *threads*. Podemos criar *threads* facilmente com Ruby utilizando a classe `Thread`:

```
1 # encoding: utf-8
2 thread = Thread.new do
3   puts "Thread #{self.object_id} iniciada!"
4   5.times do |valor|
5     puts valor
6     sleep 1
7   end
8 end
9 puts "já criei a thread"
10 thread.join
11
12 $ ruby code/thr1.rb
13 Thread 84077870 iniciada!
14 0
15 já criei a thread
16 1
17 2
18 3
19 4
```

O método `join` é especialmente útil para fazer a *thread* se completar antes que o interpretador termine. Podemos inserir um `timeout`:

```
1 # encoding: utf-8
2 thread = Thread.new do
3   puts "Thread #{self.object_id} iniciada!"
4   5.times do |valor|
5     puts valor
6     sleep 1
7   end
8 end
9 puts "já criei a thread"
10 thread.join(3)
11
12 $ ruby code/thr2.rb
13 já criei a thread
14 Thread 76000560 iniciada!
```

```
15 0
16 1
17 2
```

Podemos criar uma Proc (lembra-se delas?) e pedir que uma Thread seja criada executando o resultado da Proc, convertendo-a em um bloco (lembra-se disso também?):

```
1  proc = Proc.new do |parametro|
2    parametro.times do |valor|
3      print "[#{valor+1}/#{parametro}]"
4      sleep 0.5
5    end
6  end
7
8  thread = nil
9  5.times do |valor|
10   thread = Thread.new(valor,&proc)
11 end
12 thread.join
13 puts "Terminado!"
14
15 $ ruby code/thr3.rb
16 [1/4][1/2][1/1][1/3][2/2][2/3][2/4][3/3][3/4][4/4]Terminado!
```

Mas temos que ficar atentos à alguns pequenos detalhes. Podemos nos deparar com algumas surpresas com falta de sincronia como:

```
1  # encoding: utf-8
2  maior, menor = 0, 0
3  log = 0
4
5  t1 = Thread.new do
6    loop do
7      maior += 1
8      menor -= 1
9    end
10 end
11
12 t2 = Thread.new do
13   loop do
14     log = menor+maior
15   end
16 end
17 sleep 3
18 puts "log vale #{log}"
```

```
19
20 $ ruby code/thr4.rb
21 log vale 1
```

O problema é que não houve sincronia entre as duas *threads*, o que nos levou a resultados diferentes no log, pois não necessariamente as variáveis eram acessadas de maneira uniforme.

Podemos resolver isso usando um `Mutex`, que permite acesso exclusivo aos objetos “travados” por ele:

```
1 # encoding: utf-8
2 maior, menor = 0, 0
3 log = 0
4 mutex = Mutex.new
5   t1 = Thread.new do
6     loop do
7       mutex.synchronize do
8         maior += 1
9         menor -= 1
10      end
11    end
12  end
13
14 t2 = Thread.new do
15   loop do
16     mutex.synchronize do
17       log = menor+maior
18     end
19   end
20 end
21 sleep 3
22 puts "log vale #{log}"
23
24 $ ruby code/thr5.rb
25 log vale 0
```

Agora correu tudo como desejado! Podemos alcançar esse resultado também usando `Monitor`:

```
1 # encoding: utf-8
2 require "monitor"
3
4 maior, menor = 0, 0
5 log = 0
6 mutex = Monitor.new
7   t1 = Thread.new do
8     loop do
9       mutex.synchronize do
10        maior += 1
11        menor -= 1
12      end
13    end
14  end
15
16 t2 = Thread.new do
17   loop do
18     mutex.synchronize do
19       log = menor+maior
20     end
21   end
22 end
23
24 sleep 3
25 puts "log vale #{log}"
26
27 $ ruby code/thr6.rb
28 log vale 0
```

A diferença dos monitores é que eles podem ser uma classe pai da classe corrente, um mixin e uma extensão de um objeto em particular.

```
1 require "monitor"
2
3 class Contador1
4   attr_reader :valor
5   include MonitorMixin
6
7   def initialize
8     @valor = 0
9     super
10  end
11
12  def incrementa
13    synchronize do
14      @valor = valor + 1
```

```
15     end
16   end
17 end
18
19 class Contador2
20   attr_reader :valor
21
22   def initialize
23     @valor = 0
24   end
25
26   def incrementa
27     @valor = valor + 1
28   end
29 end
30
31 c1 = Contador1.new
32 c2 = Contador2.new
33 c2.extend(MonitorMixin)
34
35 t1 = Thread.new { 100_000.times { c1.incrementa } }
36 t2 = Thread.new { 100_000.times { c1.incrementa } }
37
38 t1.join
39 t2.join
40 puts c1.valor
41
42 t3 = Thread.new { 100_000.times { c2.synchronize { c2.incrementa } } }
43 t4 = Thread.new { 100_000.times { c2.synchronize { c2.incrementa } } }
44
45 t3.join
46 t4.join
47 puts c2.valor
48
49 $ ruby code/thr7.rb
50 200000
51 200000
```

Também para evitar a falta de sincronia, podemos ter **variáveis de condição** que sinalizam quando um recurso está ocupado ou liberado, através de `wait(mutex)` e `signal`. Vamos fazer duas threads seguindo o conceito de produtor/consumidor:

```
1 require "thread"
2
3 items = []
4 lock = Mutex.new
5 cond = ConditionVariable.new
6 limit = 0
7
8 produtor = Thread.new do
9   loop do
10    lock.synchronize do
11      qtde = rand(50)
12      next if qtde==0
13      puts "produzindo #{qtde} item(s)"
14      items = Array.new(qtde, "item")
15      cond.wait(lock)
16      puts "consumo efetuado!"
17      puts "-"*25
18      limit += 1
19    end
20    break if limit > 5
21  end
22 end
23
24 consumidor = Thread.new do
25   loop do
26     lock.synchronize do
27       if items.length>0
28         puts "consumindo #{items.length} item(s)"
29         items = []
30       end
31       cond.signal
32     end
33   end
34 end
35 produtor.join
36
37 $ ruby code/thr8.rb
38 produzindo 48 item(s)
39 consumindo 48 item(s)
40 consumo efetuado!
41 -----
42 produzindo 43 item(s)
43 consumindo 43 item(s)
44 consumo efetuado!
45 -----
46 produzindo 21 item(s)
```

```

47 consumindo 21 item(s)
48 consumo efetuado!
49 -----
50 produzindo 29 item(s)
51 consumindo 29 item(s)
52 consumo efetuado!
53 -----
54 produzindo 31 item(s)
55 consumindo 31 item(s)
56 consumo efetuado!
57 -----
58 produzindo 43 item(s)
59 consumindo 43 item(s)
60 consumo efetuado!
61 -----

```

O produtor produz os itens, avisa o consumidor que está tudo ok, o consumidor consome os itens e sinaliza para o produtor que pode enviar mais.

Comportamento similar de produtor/consumidor também pode ser alcançado utilizando Queues:

```

1  require "thread"
2
3  queue = Queue.new
4  limit = 0
5
6  produtor = Thread.new do
7    loop do
8      qtde = rand(50)
9      next if qtde==0
10     limit += 1
11     break if limit > 5
12     puts "produzindo #{qtde} item(s)"
13     queue.enq(Array.new(qtde, "item"))
14   end
15 end
16
17 consumidor = Thread.new do
18   loop do
19     obj = queue.deq
20     break if obj == :END_OF_WORK
21     print "consumindo #{obj.size} item(s)\n"
22   end
23 end
24 produtor.join
25 queue.enq(:END_OF_WORK)

```

```
26 consumidor.join
27
28 $ ruby code/thr9.rb
29 produzindo 26 item(s)
30 consumindo 26 item(s)
31 produzindo 26 item(s)
32 consumindo 26 item(s)
33 produzindo 42 item(s)
34 consumindo 42 item(s)
35 produzindo 14 item(s)
36 consumindo 14 item(s)
37 produzindo 4 item(s)
38 consumindo 4 item(s)
```

A implementação das threads das versões 1.8.x usam *green threads* e não *native threads*. As *green threads* podem ficar bloqueadas se dependentes de algum recurso do sistema operacional, como nesse exemplo, onde utilizamos um FIFO ²⁰ (o do exemplo pode ser criado em um sistema Unix-like com `mkfifo teste.fifo`) para criar o bloqueio:

```
1 proc = Proc.new do |numero|
2   loop do
3     puts "Proc #{numero}: #{'date'}"
4   end
5 end
6
7 fifo = Proc.new do
8   loop do
9     puts File.read("teste.fifo")
10  end
11 end
12
13 threads = []
14 (1..5).each do |numero|
15   threads << (numero==3 ? Thread.new(&fifo) : Thread.new(numero,&proc))
16 end
17 threads.each(&:join)
```

Podemos interceptar um comportamento “bloqueante” também utilizando o método `try_lock`. Esse método tenta bloquear o `Mutex`, e se não conseguir, retorna `false`. Vamos supor que temos uma `Thread` que efetua um processamento de tempos em tempos, e queremos verificar o resultado corrente, aproveitando para colocar um *hook* para sairmos do programa usando `CTRL+C`:

²⁰<http://pt.wikipedia.org/wiki/FIFO>

```
1  # encoding: utf-8
2  mutex = Mutex.new
3  last_result = 1
4  last_update = Time.now
5
6  trap("SIGINT") do
7    puts "saindo do programa ..."
8    exit
9  end
10
11 Thread.new do
12   loop do
13     sleep 5
14     puts "atualizando em #{Time.now} ..."
15     mutex.synchronize do
16       # alguma coisa demorada aqui
17       sleep 10
18       last_result += 1
19     end
20     last_update = Time.now
21     puts "atualizado em #{last_update}."
22   end
23 end
24
25 loop do
26   puts "aperte ENTER para ver o resultado:"
27   gets
28   if mutex.try_lock
29     begin
30       puts "resultado atualizado em #{last_update}: #{last_result}"
31     ensure
32       mutex.unlock
33     end
34   else
35     puts "sendo atualizado, resultado anterior em #{last_update}: #{last_\
36 result}"
37   end
38 end
39
40 $ ruby code/thr11.rb
41 aperte ENTER para ver o resultado:
42 resultado atualizado em 2011-07-05 18:35:54 -0300: 1
43
44 aperte ENTER para ver o resultado:
45 atualizando em 2011-07-05 18:35:59 -0300 ...
46 sendo atualizado, resultado anterior em 2011-07-05 18:35:54 -0300: 1
```

```
47
48 aperte ENTER para ver o resultado:
49 atualizado em 2011-07-05 18:36:09 -0300.
50 atualizando em 2011-07-05 18:36:14 -0300 ...
51 atualizado em 2011-07-05 18:36:24 -0300.
52 resultado atualizado em 2011-07-05 18:36:24 -0300: 3
53
54 aperte ENTER para ver o resultado:
55 resultado atualizado em 2011-07-05 18:36:24 -0300: 3
56
57 aperte ENTER para ver o resultado:
58 atualizando em 2011-07-05 18:36:29 -0300 ...
59 ^Csaindo do programa ...
```

Fibers

Entre as *features* novas do Ruby 1.9, existe uma bem interessante chamada `Fibers`, volta e meia definidas como “*threads* leves”. Vamos dar uma olhada nesse código:

```
1 3.times {|item| puts item}
```

Até aí tudo bem, aparentemente um código normal que utiliza um iterador, mas vamos dar uma olhada nesse aqui:

```
1 enum1 = 3.times
2 enum2 = %w(zero um dois).each
3 puts enum1.class
4 loop do
5     puts enum1.next
6     puts enum2.next
7 end
8
9 $ ruby code/fibers1.rb
10 Enumerator
11 0
12 zero
13 1
14 um
15 2
16 dois
```

Dando uma olhada no nome da classe de `enum1`, podemos ver que agora podemos criar um `Enumerator` com vários dos iteradores à que já estávamos acostumados, e foi o que fizemos ali alternando entre os elementos dos dois `Enumerators`, até finalizar quando foi gerada uma exceção, capturada pela estrutura `loop...do`, quando os elementos terminaram.

O segredo nos `Enumerators` é que eles estão utilizando internamente as `Fibers`. Para um exemplo básico de `Fibers`, podemos ver como calcular, novamente, os números de Fibonacci:

```
1 fib = Fiber.new do
2   x, y = 0, 1
3   loop do
4     Fiber.yield y
5     x,y = y,x+y
6   end
7 end
8 10.times { puts fib.resume }
9
10 $ ruby code/fibers2.rb
11 1
12 1
13 2
14 3
15 5
16 8
17 13
18 21
19 34
20 55
```

O segredo ali é que `Fibers` são **corrotinas** e não **subrotinas**. Em uma subrotina o controle é retornado para o contexto de onde ela foi chamada geralmente com um `return`, e continua a partir dali liberando todos os recursos alocados dentro da rotina, como variáveis locais etc.

Em uma corrotina, o controle é desviado para outro ponto mas mantendo o contexto onde ele se encontra atualmente, de modo similar à uma `closure`. O exemplo acima funciona dessa maneira:

1. A `Fiber` é criada com `new`.
2. Dentro de um iterador que vai rodar 10 vezes, é chamado o método `resume`.
3. É executado o código do início do “corpo” da `Fiber` até `yield`.
4. Nesse ponto, o controle é transferido com o valor de `y` para onde foi chamado o `resume`, e impresso na tela.
5. A partir do próximo `resume`, o código da `Fiber` é executado do ponto onde parou para baixo, ou seja, da próxima linha após o `yield` (linha 5, mostrando outra característica das corrotinas, que é ter mais de um ponto de entrada) processando os valores das variáveis e retornando para o começo do `loop`, retornando o controle novamente com `yield`.
6. Podemos comprovar que `x` e `y` tiveram seus valores preservados entre as trocas de controle.

Código parecido seria feito com uma `Proc`, dessa maneira:

```
1 def create_fib
2   x, y = 0, 1
3   lambda do
4     t, x, y = y, y, x+y
5     return t
6   end
7 end
8
9 proc = create_fib
10 10.times { puts proc.call }
11
12 $ ruby code/fibers3.rb
13 1
14 1
15 2
16 3
17 5
18 8
19 13
20 21
21 34
22 55
```

Nesse caso podemos ver o comportamento da `Proc` como uma **subrotina**, pois o valor que estamos interessados foi retornado com um `return` explícito (lembrem-se que em Ruby a última expressão avaliada é a retornada, inserimos o `return` explicitamente apenas para efeitos didáticos).

Mas ainda há algumas divergências entre `Fibers` serem corrotinas ou semi-corrotinas. As semi-corrotinas são diferentes das corrotinas pois só podem transferir o controle para quem as chamou, enquanto corrotinas podem transferir o controle para outra corrotina.

Para jogar um pouco de lenha na fogueira, vamos dar uma olhada nesse código:

```
1 f2 = Fiber.new do |value|
2   puts "Estou em f2 com #{value}, transferindo para onde vai resumir ..."
3   Fiber.yield value + 40
4   puts "Cheguei aqui?"
5 end
6
7 f1 = Fiber.new do
8   puts "Comecei f1, transferindo para f2 ..."
9   f2.resume 10
10 end
11
12 puts "Resumindo fiber 1: #{f1.resume}"
13
```

```

14 $ ruby code/fibers4.rb
15 Comecei f1, transferindo para f2 ...
16 Estou em f2 com 10, transferindo para onde vai resumir ...
17 Resumindo fiber 1: 50

```

Comportamento parecido com as semi-corrotinas! Mas e se fizermos isso:

```

1 require "fiber"
2
3 f1 = Fiber.new do |other|
4   puts "Comecei f1, transferindo para f2 ..."
5   other.transfer Fiber.current, 10
6 end
7
8 f2 = Fiber.new do |caller,value|
9   puts "Estou em f2, transferindo para f1 ..."
10  caller.transfer value + 40
11  puts "Cheguei aqui?"
12 end
13
14 puts "Resumindo fiber 1: #{f1.resume(f2)}"
15
16 $ ruby code/fibers5.rb
17 Comecei f1, transferindo para f2 ...
18 Estou em f2, transferindo para f1 ...
19 Resumindo fiber 1: 50

```

Nesse caso, f1 está transferindo o controle para f2 (que não é quem a chamou!), que transfere de volta para f1 que retorna o resultado em resume.

Discussões teóricas à parte, as Fibers são um recurso muito interessante. Para finalizar, um bate-bola rápido no esquema de “produtor-consumidor” usando Fibers:

```

1 require "fiber"
2
3 produtor = Fiber.new do |cons|
4   5.times do
5     items = Array.new((rand*5).to_i+1,"oi!")
6     puts "Produzidos #{items} ..."
7     cons.transfer Fiber.current, items
8   end
9 end
10
11 consumidor = Fiber.new do |prod,items|
12   loop do
13     puts "Consumidos #{items}"

```

```

14     prod, items = prod.transfer
15   end
16 end
17
18 produtor.resume consumidor
19
20 $ ruby code/fibers6.rb
21 Produzidos ["oi!","oi!","oi!","oi!","oi!"] ...
22 Consumidos ["oi!","oi!","oi!","oi!","oi!"]
23
24 Produzidos ["oi!","oi!","oi!","oi!","oi!"] ...
25 Consumidos ["oi!","oi!","oi!","oi!","oi!"]
26
27 Produzidos ["oi!"] ...
28 Consumidos ["oi!"]
29
30 Produzidos ["oi!","oi!","oi!","oi!","oi!"] ...
31 Consumidos ["oi!","oi!","oi!","oi!","oi!"]
32
33 Produzidos ["oi!","oi!","oi!"] ...
34 Consumidos ["oi!","oi!","oi!"]

```

As Fibers também podem ajudar a separar contextos e funcionalidades em um programa. Se precisássemos detectar a frequência de palavras em uma String ou arquivo, poderíamos utilizar uma Fiber para separar as palavras, retornando para um contador:

```

1  # encoding: utf-8
2  str = <<<FIM
3  texto para mostrar como podemos separar palavras do texto
4  para estatística de quantas vezes as palavras se repetem no
5  texto
6  FIM
7
8  scanner = Fiber.new do
9    str.scan(/\w\p{Latin}+/) do |word|
10     Fiber.yield word.downcase
11   end
12   puts "acabou!"
13 end
14
15 words = Hash.new(0)
16 while word = scanner.resume
17   words[word] += 1
18 end
19 words.each do |word,count|
20   puts "#{word}:#{count}"

```

```
21 end
22
23 $ ruby code/fibers7.rb
24 acabou!
25 texto:3
26 para:2
27 mostrar:1
28 como:1
29 podemos:1
30 separar:1
31 palavras:2
32 do:1
33 estatística:1
34 de:1
35 quantas:1
36 vezes:1
37 as:1
38 se:1
39 repetem:1
40 no:1
```



Dica

Estão vendo como eu escrevi a expressão regular acima? O `\p{Latin}` é uma propriedade de caracter que habilita a expressão regular a entender os nossos caracteres acentuados.

Mais sobre as propriedades de caracteres na [documentação do Ruby](#)²¹.



Desafio 5

Tente fazer a frequência das palavras utilizando iteradores e blocos.

Fica uma dica que dá para fazer utilizando a mesma expressão regular e uma Hash.

²¹<http://www.ruby-doc.org/core-1.9.3/Regexp.html#label-Character+Properties>

Continuations

Ruby também tem suporte à Continuations, que são, segundo a *Wikipedia*²²,

“Representações abstratas do controle de estado de um programa”

Um exemplo nos mostra que a *call stack* de um programa é preservada chamando uma Continuation:

```
1 require "continuation"
2
3 def cria_continuation
4   puts "Criando a continuation e retornando ..."
5   callcc {|obj| return obj}
6   puts "Ei, olha eu aqui de volta na continuation!"
7 end
8
9 puts "Vou criar a continuation."
10 cont = cria_continuation()
11 puts "Verificando se existe ..."
12
13 if cont
14   puts "Criada, vamos voltar para ela?"
15   cont.call
16 else
17   puts "Agora vamos embora."
18 end
19 puts "Terminei, tchau."
20
21 $ ruby code/cont.rb
22 Vou criar a continuation.
23 Criando a continuation e retornando ...
24 Verificando se existe ...
25 Criada, vamos voltar para ela?
26 Ei, olha eu aqui de volta na continuation!
27 Verificando se existe ...
28 Agora vamos embora.
29 Terminei, tchau.
```

²²<http://en.wikipedia.org/wiki/Continuations>

Processos em paralelo

Podemos utilizar a *gem* `Parallel`²³ para executar processamento em paralelo usando processos (em CPUs com vários processadores) ou utilizando as `Threads`:

```
1 gem install parallel
```

Vamos ver um exemplo utilizando `Threads`, que dão mais velocidade em operações bloqueantes, não usam memória extra e permitem modificação de dados globais:

```
1 require "parallel"
2
3 puts Time.now
4 res = "Quem terminou primeiro? "
5 Parallel.map 1..20, :in_threads => 4 do |nr|
6   5.times {|t| sleep rand; print "'#{nr}/#{t}' " }
7   puts "acabei com #{nr} "
8   res += "'#{nr} "
9 end
10 puts res
11 puts Time.now
12
13 $ ruby par.rb
14 2011-07-08 14:56:43 -0300
15 '4/0' '3/0' '3/1' '2/0' '4/1' '1/0' '2/1' '1/1' '3/2' '4/2' '1/2' '4/3' '3/3'
16 '1/3' '2/2' '3/4' acabei com 3
17 '4/4' acabei com 4
18 '5/0' '2/3' '6/0' '1/4' acabei com 1
19 '5/1' '2/4' acabei com 2
20 '7/0' '5/2' '5/3' '6/1' '6/2' '6/3' '5/4' acabei com 5
21 '8/0' '7/1' '8/1' '9/0' '7/2' '7/3' '6/4' acabei com 6
22 '8/2' '9/1' '9/2' '10/0' '9/3' '7/4' acabei com 7
23 '8/3' '9/4' acabei com 9
24 '11/0' '10/1' '8/4' acabei com 8
25 '10/2' '13/0' '10/3' '10/4' acabei com 10
26 '12/0' '11/1' '13/1' '13/2' '12/1' '14/0' '11/2' '11/3' '13/3' '12/2' '14/1'
27 '14/2' '12/3' '14/3' '13/4' acabei com 13
28 '11/4' acabei com 11
29 '12/4' acabei com 12
30 '15/0' '14/4' acabei com 14
```

²³<https://github.com/grosser/parallel>

```

31 '18/0' '17/0' '15/1' '18/1' '16/0' '16/1' '16/2' '16/3' '17/1' '18/2' '15/2'
32 '16/4' acabei com 16
33 '15/3' '17/2' '19/0' '18/3' '17/3' '19/1' '18/4' acabei com 18
34 '15/4' acabei com 15
35 '20/0' '19/2' '17/4' acabei com 17
36 '19/3' '20/1' '19/4' acabei com 19
37 '20/2' '20/3' '20/4' acabei com 20
38
39 Quem terminou primeiro? 3 4 1 2 5 6 7 9 8 10 13 11 12 14 16 18 15 17 19 20
40 2011-07-08 14:56:57 -0300

```

Agora, utilizando processos, que utilizam mais de um núcleo, dão mais velocidade para operações bloqueantes, protegem os dados globais, usam mais alguma memória e permitem interromper os processos filhos junto com o processo principal, através de CTRL+C ou enviando um sinal com `kill -2`:

```

1 require "parallel"
2
3 puts Time.now
4 res = "Quem terminou primeiro? "
5 Parallel.map 1..20, :in_processes => 3 do |nr|
6   5.times {|t| sleep rand; print "'#{nr}/#{t}' " }
7   puts "acabei com #{nr} "
8   res += "'#{nr} "
9 end
10 puts res
11 puts Time.now
12
13 $ ruby par2.rb
14 2011-07-08 15:03:32 -0300
15 '3/0' '1/0' '2/0' '3/1' '2/1' '1/1' '3/2' '2/2' '2/3' '1/2' '3/3' '2/4' acabe\
16 i com 2
17 '1/3' '3/4' acabei com 3
18 '5/0' '4/0' '1/4' acabei com 1
19 '5/1' '4/1' '6/0' '5/2' '4/2' '6/1' '6/2' '5/3' '4/3' '5/4' acabei com 5
20 '4/4' acabei com 4
21 '6/3' '8/0' '8/1' '7/0' '6/4' acabei com 6
22 '8/2' '9/0' '9/1' '7/1' '8/3' '9/2' '8/4' acabei com 8
23 '9/3' '7/2' '7/3' '7/4' acabei com 7
24 '9/4' acabei com 9
25 '12/0' '10/0' '12/1' '11/0' '10/1' '12/2' '12/3' '12/4' acabei com 12
26 '11/1' '13/0' '10/2' '13/1' '11/2' '10/3' '13/2' '11/3' '11/4' acabei com 11
27 '10/4' acabei com 10
28 '15/0' '13/3' '15/1' '13/4'
29 '14/0' '14/1' '15/2' '15/3'
30 '16/1' '17/0' '16/2' '14/4'

```

```

31 '17/1' '18/0' '16/3' '17/2'
32 '18/2' '19/0' '17/3' '17/4'
33 '19/1' '19/2' '18/3' '19/3'
34 '18/4' acabei com 18
35 '20/1' '20/2' '20/3' '20/4'
36
37 Quem terminou primeiro?
38 2011-07-08 15:03:50 -0300

```



Desafio 4 Tente descobrir a diferença entre o código que utilizou threads e processes

Para executar esse mesmo código utilizando o número de processadores da CPU, é só não especificar nem `in_threads` ou `in_processes`:

```

1 require "parallel"
2
3 puts Time.now
4 res = "Quem terminou primeiro? "
5 Parallel.map 1..20 do |nr|
6   5.times {|t| sleep rand; print "'#{nr}/#{t}' " }
7   puts "acabei com #{nr} "
8   res += "'#{nr} "
9 end
10 puts res
11 puts Time.now
12
13 $ ruby par3.rb
14 2011-07-08 15:07:05 -0300
15 '1/0' '2/0' '1/1' '2/1' '1/2' '2/2' '1/3' '2/3' '1/4' acabei com 1
16 '2/4' acabei com 2
17 '3/0' '4/0' '4/1' '3/1' '3/2' '4/2' '4/3' '4/4' acabei com 4
18 '3/3' '3/4' acabei com 3
19 '5/0' '6/0' '5/1' '5/2' '6/1' '5/3' '6/2' '5/4' acabei com 5
20 '6/3' '7/0' '7/1' '7/2' '6/4' acabei com 6
21 '8/0' '7/3' '8/1' '7/4' acabei com 7
22 '8/2' '9/0' '8/3' '9/1' '9/2' '8/4' acabei com 8
23 '10/0' '9/3' '10/1' '9/4' acabei com 9
24 '10/2' '11/0' '11/1' '10/3' '11/2' '11/3' '11/4' acabei com 11
25 '10/4' acabei com 10
26 '13/0' '12/0' '13/1' '13/2' '12/1' '13/3' '13/4' acabei com 13
27 '12/2' '12/3' '12/4' acabei com 12
28 '14/0' '15/0' '15/1' '14/1' '14/2' '14/3' '15/2' '14/4' acabei com 14
29 '16/0' '15/3' '16/1' '15/4' acabei com 15

```

```

30 '16/2' '17/0' '16/3' '17/1' '16/4' acabei com 16
31 '17/2' '17/3' '17/4' acabei com 17
32 '19/0' '19/1' '18/0' '19/2' '18/1' '19/3' '18/2' '18/3' '18/4' acabei com 18
33 '19/4' acabei com 19
34 '20/0' '20/1' '20/2' '20/3' '20/4' acabei com 20
35 Quem terminou primeiro?
36 2011-07-08 15:07:34 -0300

```

Fazendo uma comparação com Threads:

```

1 puts Time.now
2 res = "Quem terminou primeiro? "
3
4 threads = []
5 (1..20).each do |nr|
6   threads << Thread.new do
7     5.times {|t| sleep rand; print "'#{nr}/#{t}' " }
8     puts "acabei com #{nr} "
9     res += "'#{nr} "
10  end
11 end
12 threads.each(&:join)
13 puts res
14 puts Time.now
15
16 $ ruby par4.rb
17 2011-07-08 11:11:29 -0300
18 '17/0' '1/0' '15/0' '2/0' '15/1' '8/0' '18/0' '7/0' '17/1' '19/0' '14/0'
19 '17/2' '10/0' '5/0' '2/1' '4/0' '9/0' '6/0' '8/1' '2/2' '5/1' '15/2' '12/0'
20 '4/1' '16/0' '11/0' '14/1' '20/0' '16/1' '13/0' '4/2' '3/0' '10/1' '19/1'
21 '20/1' '10/2' '7/1' '18/1' '13/1' '18/2' '1/1' '14/2' '1/2' '17/3' '14/3'
22 '8/2' '6/1' '12/1' '4/3' '6/2' '4/4' acabei com 4
23 '15/3' '17/4' acabei com 17
24 '5/2' '11/1' '9/1' '16/2' '2/3' '7/2' '14/4' acabei com 14
25 '12/2' '19/2' '3/1' '18/3' '13/2' '10/3' '7/3' '20/2' '1/3' '10/4' acabei com\
26 10
27 '15/4' acabei com 15
28 '6/3' '8/3' '5/3' '5/4' acabei com 5
29 '6/4' acabei com 6
30 '20/3' '7/4' acabei com 7
31 '11/2' '13/3' '12/3' '13/4' acabei com 13
32 '2/4' acabei com 2
33 '19/3' '9/2' '19/4' acabei com 19
34 '11/3' '3/2' '16/3' '1/4' acabei com 1
35 '18/4' acabei com 18
36 '20/4' acabei com 20

```

```
37 '8/4' acabei com 8
38 '11/4' acabei com 11
39 '9/3' '12/4' acabei com 12
40 '9/4' acabei com 9
41 '16/4' acabei com 16
42 '3/3' '3/4' acabei com 3
43 Quem terminou primeiro? 4 17 14 10 15 5 6 7 13 2 19 1 18 20 8 11 12 9 16 3
44 2011-07-08 11:11:32 -0300
```

Benchmarks

Ao invés de medir nosso código através do sucessivas chamadas à `Time.now`, podemos utilizar o módulo de *benchmark*, primeiro medindo uma operação simples, como criar uma `String`:

```
1 require "benchmark"
2 Benchmark.measure { "-"*1_000_000 }
3 =>  0.000000  0.000000  0.000000  0.002246
```

Ou um pedaço de código:

```
1 require "benchmark"
2 require "parallel"
3
4 Benchmark.bm do |bm|
5   bm.report do
6     Parallel.map 1..20, :in_threads => 4 do |nr|
7       5.times {|t| sleep rand; }
8     end
9   end
10 end
11
12 $ ruby bench1.rb
13   user      system      total      real
14 0.040000   0.030000   0.070000 ( 13.937973)
```

Podemos comparar vários pedaços de código, dando uma *label* para cada um:

```
1 # encoding: utf-8
2 require "benchmark"
3 require "parallel"
4
5 Benchmark.bm do |bm|
6   bm.report("in_threads") do
7     Parallel.map 1..20, :in_threads => 4 do |nr|
8       5.times {|t| sleep 0.5; }
9     end
10  end
11  bm.report("in_processes") do
12    Parallel.map 1..20, :in_processes => 4 do |nr|
13      5.times {|t| sleep 0.5; }
```

```
14     end
15   end
16   bm.report("using threads") do
17     threads = []
18     (1..20).each do |nr|
19       threads << Thread.new do
20         5.times {|t| sleep 0.5; }
21       end
22     end
23     threads.each(&:join)
24   end
25 end
26
27 $ ruby bench2.rb
28           user  system  total      real
29 in_threads:  0.030000 0.030000 0.060000 ( 12.277710)
30 in_processes: 0.000000 0.060000 0.240000 ( 17.514098)
31 using threads: 0.010000 0.000000 0.010000 (  3.303277)
```

Entrada e saída

Ler, escrever e processar arquivos e fluxos de rede são requisitos fundamentais para uma boa linguagem de programação moderna. Em algumas, apesar de contarem com vários recursos para isso, às vezes são muito complicados ou burocráticos, o que com tantas opções e complexidade várias vezes pode confundir o programador. Em Ruby, como tudo o que vimos até aqui, vamos ter vários meios de lidar com isso de forma descomplicada e simples.

Arquivos

Antes de começarmos a lidar com arquivos, vamos criar um arquivo novo para fazermos testes, com o nome criativo de `teste.txt`. Abra o seu editor de texto (pelo amor, eu disse **editor** e não **processador** de textos, a cada vez que você confunde isso e abre o Word alguém solta um pum no elevador) e insira o seguinte conteúdo:

```
1 Arquivo de teste
2 Curso de Ruby
3 Estamos na terceira linha.
4 E aqui é a quarta e última.
```

Podemos ler o arquivo facilmente, utilizando a classe `File` e o método `read`:

```
1 p File.read("teste.txt")
2
3 $ ruby code/io1.rb
4 "Arquivo de teste\nCurso de Ruby\nEstamos na terceira linha.\nE aqui é a quar\
5 ta e última.\n"
```

Isso gera uma `String` com todo o conteúdo do arquivo, porém sem a quebra de linhas presente no arquivo. Para lermos todas as suas linhas como um `Array` (que teria o mesmo efeito de quebrar a `String` resultante da operação acima em `\n`):

```
1 p File.readlines("teste.txt")
2
3 $ ruby code/io2.rb
4 ["Arquivo de teste\n", "Curso de Ruby\n", "Estamos na terceira linha.\n", "E \
5 aqui é a quarta e última.\n"]
```

Podemos abrir o arquivo especificando o seu modo e armazenando o seu *handle*. O modo para leitura é `r` e para escrita é `w`. Podemos usar o iterador do *handle* para ler linha a linha:

```

1 f = File.open("teste.txt")
2 f.each do |linha|
3   puts linha
4 end
5 f.close
6
7 $ ruby code/io3.rb
8 Arquivo de teste
9 Curso de Ruby
10 Estamos na terceira linha.
11 E aqui é a quarta e última.

```

Melhor do que isso é passar um bloco para `File` onde o arquivo vai ser aberto e automaticamente fechado no final do bloco:

```

1 File.open("teste.txt") do |arquivo|
2   arquivo.each do |linha|
3     puts linha
4   end
5 end

```

Isso “automagicamente” vai fechar o *handle* do arquivo, no final do bloco.

Para ler o arquivo *byte a byte*, podemos fazer:

```

1 File.open("teste.txt") do |arquivo|
2   arquivo.each_byte do |byte|
3     print "[#{byte}]"
4   end
5 end
6
7 $ ruby code/io5.rb
8 [65][114][113][117][105][118][111][32][100][101][32][116][101][115][116]
9 [101][10][67][117][114][115][111][32][100][101][32][82][117][98][121]
10 [10][69][115][116][97][109][111][115][32][110][97][32][116][101][114]
11 [99][101][105][114][97][32][108][105][110][104][97][46][10][69][32][97]
12 [113][117][105][32][195][169][32][97][32][113][117][97][114][116][97]
13 [32][101][32][195][186][108][116][105][109][97][46][10]

```

Para escrever em um arquivo, fazendo uma cópia do atual:

```

1 File.open("novo_teste.txt", "w") do |arquivo|
2   arquivo << File.read("teste.txt")
3 end

```

Arquivos Zip

Podemos ler e escrever em arquivos compactados Zip, para isso vamos precisar da *gem* `rubyzip`:

```
1 gem install rubyzip
```

Vamos criar três arquivos, 1.txt, 2.txt e 3.txt com conteúdo livre dentro de cada um, que vão ser armazenados internamente no arquivo em um subdiretório chamado txts, compactando e logo descompactando:

```
1 require "rubygems"
2 require "zip"
3 require "fileutils"
4
5 myzip = "teste.zip"
6 File.delete(myzip) if File.exists?(myzip)
7
8 Zip::File.open(myzip,true) do |zipfile|
9   Dir.glob("[0-9]*.txt") do |file|
10    puts "Zipando #{file}"
11    zipfile.add("txts/#{file}",file)
12  end
13 end
14
15 Zip::File.open(myzip) do |zipfile|
16   zipfile.each do |file|
17     dir = File.dirname(file.name)
18     puts "Descompactando #{file.name} para #{dir}"
19     FileUtils.mkpath(dir) if !File.exists?(dir)
20     zipfile.extract(file.name,file.name) do |entry,file|
21       puts "Arquivo #{file} existe, apagando ..."
22       File.delete(file)
23     end
24   end
25 end
```

Rodando o programa:

```
1 $ ruby code/io7.rb
2 Zipando 3.txt
3 Zipando 1.txt
4 Zipando 2.txt
5 Descompactando txts/3.txt para txts
6 Descompactando txts/1.txt para txts
7 Descompactando txts/2.txt para txts
8 $ ls txts
9 total 20K
10 drwxr-xr-x 2 taq taq 4,0K 2011-07-06 15:16 .
11 drwxr-xr-x 6 taq taq 4,0K 2011-07-06 15:16 ..
```

```
12 -rw-r--r-- 1 taq taq 930 2011-07-06 15:16 1.txt
13 -rw-r--r-- 1 taq taq 930 2011-07-06 15:16 2.txt
14 -rw-r--r-- 1 taq taq 930 2011-07-06 15:16 3.txt
```

Algumas explicações sobre o código:

- Na linha 3 foi requisitado o módulo `FileUtils`, que carrega métodos como o `mkpath`, na linha 19, utilizado para criar o diretório (ou a estrutura de diretórios).
- Na linha 8 abrimos o arquivo, enviando `true` como *flag* indicando para criar o arquivo caso não exista. Para arquivos novos, podemos também utilizar `new`.
- Na linha 9 utilizamos `Dir.glob` para nos retornar uma lista de arquivos através de uma máscara de arquivos.
- Na linha 11 utilizamos o método `add` para inserir o arquivo encontrado dentro de um *path* interno do arquivo compactado, nesse caso dentro de um diretório chamado `txts`.
- Na linha 15 abrimos o arquivo criado anteriormente, para leitura.
- Na linha 16 utilizamos o iterador `each` para percorrer os arquivos contidos dentro do arquivo compactado.
- Na linha 17 extraímos o nome do diretório com `dirname`.
- Na linha 20 extraímos o arquivo, passando um bloco que vai ser executado no caso do arquivo já existir.

XML

Vamos acessar arquivos XML através do REXML, um processador XML que já vem com Ruby. Para mais informações sobre esse processador XML, consulte o tutorial oficial em <http://www.germane-software.com/software/rexml/docs/tutorial.html>²⁴.

Antes de mais nada, vamos criar um arquivo XML para os nossos testes, chamado `aluno.xml`, usando o REXML para isso:

```
1 # encoding: utf-8
2 require "rexml/document"
3
4 doc = REXML::Document.new
5 decl = REXML::XMLDecl.new("1.0", "UTF-8")
6 doc.add decl
7
8 root = REXML::Element.new("alunos")
9 doc.add_element root
10
11 alunos = [[1, "João"], [2, "José"], [3, "Antonio"], [4, "Maria"]]
12 alunos.each do |info|
13   aluno = REXML::Element.new("aluno")
```

²⁴<http://www.germane-software.com/software/rexml/docs/tutorial.html>

```
14     id     = REXML::Element.new("id")
15     nome  = REXML::Element.new("nome")
16
17     id.text  = info[0]
18     nome.text = info[1]
19
20     aluno.add_element id
21     aluno.add_element nome
22     root.add_element aluno
23 end
24 doc.write(File.open("alunos.xml", "w"))
```

O resultado será algo como:

```
1 $ cat alunos.xml
2 <?xml version='1.0' encoding='UTF-8'?>
3 <alunos>
4   <aluno>
5     <id>1</id>
6     <nome>João</nome>
7   </aluno>
8   <aluno>
9     <id>2</id>
10    <nome>José</nome>
11  </aluno>
12  <aluno>
13    <id>3</id>
14    <nome>Antonio</nome>
15  </aluno>
16  <aluno>
17    <id>4</id>
18    <nome>Maria</nome>
19  </aluno>
20 </alunos>
```

Agora vamos ler esse arquivo. Vamos supor que eu quero listar os dados de todos os alunos:

```
1 require "rexml/document"
2
3 doc = REXML::Document.new(File.open("alunos.xml"))
4 doc.elements.each("alunos/aluno") do |aluno|
5     puts "#{aluno.elements['id'].text}-#{aluno.elements['nome'].text}"
6 end
7
8 $ ruby code/xml2.rb
9 1-João
10 2-José
11 3-Antonio
12 4-Maria
```

Poderíamos ter convertido também os elementos em um Array e usado o iterador para percorrer o arquivo, o que dará resultado similar:

```
1 require "rexml/document"
2
3 doc = REXML::Document.new(File.open("alunos.xml"))
4 doc.elements.to_a("//aluno").each do |aluno|
5     puts "#{aluno.elements['id'].text}-#{aluno.elements['nome'].text}"
6 end
```

Se quiséssemos somente o segundo aluno, poderíamos usar:

```
1 require "rexml/document"
2
3 doc = REXML::Document.new(File.open("alunos.xml"))
4 root = doc.root
5 aluno = root.elements["aluno[2]"]
6 puts "#{aluno.elements['id'].text}-#{aluno.elements["nome"].text}"
7
8 $ ruby code/xml4.rb
9 2-José
```

Uma abordagem mais moderna para criar XML em Ruby é a *gem* builder:

```
1 $ gem install builder
2
3 # encoding: utf-8
4 require "builder"
5
6 alunos = {1=>"João",2=>"José",3=>"Antonio",4=>"Maria"}
7
8 xml = Builder::XmlMarkup.new(:indent=>2)
9 xml.alunos do
10   alunos.each do |key,value|
11     xml.aluno do
12       xml.id key
13       xml.nome value
14     end
15   end
16 end
17
18 # para gravar o arquivo
19 File.open("alunos.xml","w") do |file|
20   file << xml.target!
21 end
22
23 $ ruby code/xml5.rb
24 <alunos>
25 <aluno>
26 <id>1</id>
27 <nome>João</nome>
28 </aluno>
29 ...
```

E para a leitura de arquivos XML, podemos utilizar a *gem* nokogiri:

```
1 $ gem install nokogiri
2
3 require "nokogiri"
4
5 doc = Nokogiri::XML(File.open("alunos.xml"))
6 doc.search("aluno").each do |node|
7   puts node.search("id").text+": "+node.search("nome").text
8 end
9
10 $ ruby code/xml6.rb
11 1:João
12 2:José
13 3:Antonio
14 4:Maria
```

XSLT

Aproveitando que estamos falando de XML, vamos ver como utilizar o XSLT. XSLT é uma linguagem para transformar documentos XML em outros documentos, sejam eles outros XML, HTML, o tipo que você quiser e puder imaginar.

XSLT é desenhado para uso com XSL, que são folhas de estilo para documentos XML. Alguns o acham muito “verboso” (sim, existe essa palavra), mas para o que ele é proposto, é bem útil. Você pode conhecer mais sobre XSLT na URL oficial do W3C ²⁵.

O uso de XSLT em Ruby pode ser feito com o uso da *gem* `ruby-xslt`:

```
1 $ gem install ruby-xslt
```

Após isso vamos usar o nosso arquivo `alunos.xml` criado anteriormente para mostrar um exemplo de transformação. Para isso vamos precisar de uma folha de estilo XSL, `alunos.xsl`:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0\
3 ">
4 <xsl:output method="html" encoding="utf-8" indent="no"/>
5   <xsl:template match="/alunos">
6     <html>
7       <head>
8         <title>Teste de XSLT</title>
9       </head>
10      <body>
11        <table>
12          <caption>Alunos</caption>
13          <thead>
14            <th>
15              <td>Id</td>
16              <td>Nome</td>
17            </th>
18          </thead>
19          <tbody>
20            <xsl:apply-templates/>
21          </tbody>
22        </table>
23      </body>
24    </html>
25  </xsl:template>
26
27  <xsl:template match="aluno">
```

²⁵<http://www.w3.org/TR/xslt>

```
28     <tr>
29         <td><xsl:value-of select="id"/></td>
30         <td><xsl:value-of select="nome"/></td>
31     </tr>
32 </xsl:template>
33 </xsl:stylesheet>
```

Agora o código Ruby:

```
1 require "xml/xslt"
2
3 xslt = XML::XSLT.new
4 xslt.xsl = "alunos.xsl"
5 xslt.xml = "alunos.xml"
6 xslt.save("alunos.html")
7 puts xslt.serve
```

Rodando o programa vamos ter o resultado gravado no arquivo `alunos.html` e apresentado na tela. Abrindo o arquivo vamos ver:

```
1 $ ruby xslt.rb | lynx --stdin
2 CAPTION: Alunos
3 Id Nome
4 1 João
5 2 José
6 3 Antonio
7 4 Maria
```



O `lynx` é um navegador em modo texto que quebra um bom galho.

JSON

Aproveitando que estamos falando de XML, nada melhor do que comparar com a alternativa mais do que otimizada utilizada largamente hoje em dia na web para transmissão de dados sem utilizar os “monstrinhos” de XML: JSON²⁶. Não é aquele cara do “Sexta-Feira 13” não hein! É o *JavaScript Object Notation*, que nos permite converter, por exemplo, uma Hash em uma String que pode ser enviada nesse formato:

²⁶<http://www.json.org/>

```

1 require "json"
2 => true
3 > {joao: 1, jose: 2, antonio: 3, maria: 4}.to_json
4 => '{"joao":1,"jose":2,"antonio":3,"maria":4}'

```

e a conversão de volta:

```

1 JSON.parse({joao: 1, jose: 2, antonio: 3, maria: 4}.to_json)
2 => {"joao"=>1, "jose"=>2, "antonio"=>3, "maria"=>4}

```

YAML

Podemos definir o YAML (YAML Ain't Markup Language - pronuncia-se mais ou menos como “ieimel”, fazendo rima com a pronúncia de “camel”, em inglês) como uma linguagem de definição ou *markup* menos verbosa que o XML.

Vamos dar uma olhada em como ler arquivos YAML convertendo-os em tipos do Ruby. Primeiro vamos criar um arquivo chamado teste.yml (a extensão dos arquivos YAML é yml) que vamos alterar de acordo com nossos exemplos, armazenando um Array no nosso arquivo.

Insira o seguinte conteúdo, lembrando que -- indica o começo de um arquivo YAML:

```

1 ---
2 - José
3 - João
4 - antonio
5 - maria

```

E agora vamos ler esse arquivo, tendo o resultado convertido em um Array:

```

1 require "yaml"
2
3 result = YAML::load(File.open(ARGV[0]))
4 p result
5
6 $ ruby leryaml.rb teste.yml
7 ["José", "João", "antonio", "maria"]

```

Podemos ter Arrays dentro de Arrays:

```
1 ---
2 -
3   - joãoo
4   - josé
5 -
6   - maria
7   - antonio
8
9 $ ruby leryaml.rb teste2.yml
10 [{"joãoo", "josé"}, ["maria", "antonio"]]
```

Agora vamos ver como fazer uma Hash:

```
1 ---
2 josé: 1
3 joãoo: 2
4 antonio: 3
5 maria: 4
6
7 $ ruby leryaml.rb teste3.yml
8 {"josé"=>1, "joãoo"=>2, "antonio"=>3, "maria"=>4}
```

Hashes dentro de Hashes:

```
1 ---
2 pessoas:
3   joãoo: 1
4   josé: 2
5   maria: 3
6   antonio: 4
7
8 $ ruby leryaml.rb teste4.yml
9 {"pessoas"=>{"joãoo"=>1, "josé"=>2, "maria"=>3, "antonio"=>4}}
```

O que nos dá, com um arquivo de configuração do banco de dados do Rails:

```
1 ---
2 development:
3   adapter: mysql
4   database: teste_development
5   username: root
6   password: test
7   host: localhost
8
9 test:
10  adapter: mysql
11  database: teste_test
12  username: root
13  password: test
14  host: localhost
15
16 production:
17  adapter: mysql
18  database: teste_production
19  username: root
20  password: test
21  host: localhost
22
23 $ ruby leryaml.rb teste5.yml
24 {"development"=>{"adapter"=>"mysql", "database"=>"teste_development",
25 "username"=>"root", "password"=>"test", "host"=>"localhost"},
26 "test"=>{"adapter"=>"mysql", "database"=>"teste_test", "username"=>"root",
27 "password"=>"test", "host"=>"localhost"}, "production"=>{"adapter"=>"mysql",
28 "database"=>"teste_production", "username"=>"root", "password"=>"test",
29 "host"=>"localhost"}}
```

TCP

O TCP é um dos protocolos que nos permitem utilizar a Internet e que define grande parte do seu funcionamento. Falar em utilizar comunicação de rede sem utilizar TCP hoje em dia é quase uma impossibilidade para grande parte das aplicações que utilizamos e que pretendemos construir. Outra vantagem é a quantidade e qualidade de documentação que podemos encontrar sobre o assunto, o que, alguns anos antes, quando alguns protocolos como o IPX/SPX e o X25 dominam respectivamente na parte de redes de computadores e transmissão telefônica, era uma tarefa bem complicada, principalmente pelo fato de não haver nem Internet para consultarmos algo. Lembro que demorei tanto para arrumar um livro decente sobre IPX/SPX que 1 ano depois, nem precisava mais dele (e não sei para onde diabos que ele foi).

Para começar a aprender sobre como utilizar TCP em Ruby, vamos verificar um servidor SMTP, usando sockets TCP, abrindo a URL indicada na porta 25:

```

1  require "socket"
2
3  TCPSocket.open("smtp.mail.yahoo.com",25) do |smtp|
4      puts smtp.gets
5      smtp.puts "EHLO bluefish.com.br"
6      puts smtp.gets
7  end
8
9  $ ruby sock.rb
10 220 smtp209.mail.ne1.yahoo.com ESMTP
11 250-smtp209.mail.ne1.yahoo.com

```

Agora vamos criar um servidor com TCP novinho em folha, na porta 8081, do localhost (quem não souber o que é localhost arrume uma ferramenta de ataque com algum script kiddie e aponte para esse tal de localhost - dependendo do seu sistema operacional e configurações de segurança dele, vai aprender rapidinho) ²⁷:

```

1  # encoding: utf-8
2  require "socket"
3
4  TCPServer.open("localhost",8081) do |server|
5      puts "servidor iniciado"
6      loop do
7          puts "aguardando conexão ..."
8          con = server.accept
9          puts "conexão recebida!"
10         con.puts Time.now
11         con.close
12     end
13 end
14
15 $ ruby tcpserver.rb
16 servidor iniciado
17 aguardando conexão ...
18 conexão recebida!
19 $ telnet localhost 8081
20 Trying ::1...
21 Connected to localhost.localdomain.
22 Escape character is '^]'.
23 2011-07-06 18:42:48 -0300
24 Connection closed by foreign host.

```

Podemos trafegar, além de Strings, outros tipos pela conexão TCP, fazendo uso dos métodos pack, para “empacotar” e unpack, para “desempacotar” os dados que queremos transmitir. Primeiro, com o arquivo do servidor, tcpserver2.rb:

²⁷<https://gist.github.com/taq/5793430>

```
1 # encoding: utf-8
2 require "socket"
3
4 TCPServer.open("localhost",8081) do |server|
5   puts "servidor iniciado"
6   loop do
7     puts "aguardando conexão ..."
8     con = server.accept
9     rst = con.recv(1024).unpack("LA10A*")
10    fix = rst[0]
11    str = rst[1]
12
13    hash = Marshal.load(rst[2])
14    puts "#{fix.class}\t: #{fix}"
15    puts "#{str.class}\t: #{str}"
16    puts "#{hash.class}\t: #{hash}"
17    con.close
18  end
19 end
```

E agora com o arquivo do cliente, tcpclient.rb:

```
1 require "socket"
2
3 hash = {um: 1,dois: 2, tres: 3}
4 TCPSocket.open("localhost",8081) do |server|
5   server.write [1,"teste".ljust(10),Marshal.dump(hash)].pack("LA10A*")
6 end
7
8 $ ruby tcpserver2.rb
9 servidor iniciado
10 aguardando conexão ...
11 Fixnum : 1
12 String : teste
13 Hash : {:um=>1, :dois=>2, :tres=>3}
14 aguardando conexão ...
15
16 $ ruby tcpclient.rb
```



Desafio 6

Você consegue descobrir o que significa aquele "LA10A*" que foi utilizado?

UDP

O protocolo UDP ²⁸ utiliza pacotes com um datagrama encapsulado que não tem a garantia que vai chegar ao seu destino, ou seja, não é confiável para operações críticas ou que necessitem de alguma garantia de entrega dos dados, mas pode ser uma escolha viável por causa da sua velocidade, a não necessidade de manter um estado da conexão e algumas outras que quem está desenvolvendo algum programa para comunicação de rede vai conhecer e levar em conta.

Vamos escrever dois programas que nos permitem enviar e receber pacotes usando esse protocolo. Primeiro, o código do servidor:

```
1 # encoding: utf-8
2 require "socket"
3
4 server = UDPSocket.new
5 porta = 12345
6 server.bind("localhost",porta)
7 puts "Servidor conectado na porta #{porta}, aguardando ..."
8 loop do
9   msg,sender = server.recvfrom(256)
10  host = sender[3]
11  puts "Host #{host} enviou um pacote UDP: #{msg}"
12  break unless msg.chomp != "kill"
13 end
14 puts "Kill recebido, fechando servidor."
15 server.close
```

Agora o código do cliente:

```
1 # encoding: utf-8
2 require "socket"
3
4 client = UDPSocket.open
5 client.connect("localhost",12345)
6 loop do
7   puts "Digite sua mensagem (quit termina, kill finaliza servidor):"
8   msg = gets
9   client.send(msg,0)
10  break unless !"kill,quit".include? msg.chomp
11 end
12 client.close
```

Rodando o servidor e o cliente:

²⁸http://pt.wikipedia.org/wiki/Protocolo_UDP

```
1 $ ruby udpserver.rb
2 Servidor conectado na porta 12345, aguardando ...
3 Host 127.0.0.1 enviou um pacote UDP: oi
4 Host 127.0.0.1 enviou um pacote UDP: tudo bem?
5 Host 127.0.0.1 enviou um pacote UDP: kill
6 Kill recebido, fechando servidor.
7
8 $ ruby code/udpclient.rb
9 Digite sua mensagem (quit termina, kill finaliza servidor):
10 oi
11
12 Digite sua mensagem (quit termina, kill finaliza servidor):
13 tudo bem?
14
15 Digite sua mensagem (quit termina, kill finaliza servidor):
16 kill
```



Dica

No método `send` o argumento 0 é uma *flag* que pode usar uma combinação T> de constantes (utilizando um or binário das constantes presentes em `Socket::MSG_*`).

SMTP

O SMTP é um protocolo para o **envio** de emails, baseado em texto. Há uma classe SMTP pronta para o uso em Ruby:

```
1 # encoding: utf-8
2 require "net/smtp"
3 require "highline/import"
4
5 from = "eustaquiorangel@gmail.com"
6 pass = ask("digite sua senha:") {|q| q.echo=""}
7 to   = "eustaquiorangel@gmail.com"
8
9 msg = <<<FIM
10 From: #{from}
11 Subject: Teste de SMTP no Ruby
12 Apenas um teste de envio de email no Ruby.
13 Falou!
14 FIM
15
16 smtp = Net::SMTP.new("smtp.gmail.com", 587)
17 smtp.enable_starttls
```

```
18
19 begin
20     smtp.start("localhost", from, pass, :plain) do |smtp|
21         puts "conexão aberta!"
22         smtp.send_message(msg, from, to)
23         puts "mensagem enviada!"
24     end
25 rescue => exception
26     puts "ERRO: #{exception}"
27     puts exception.backtrace
28 end
29
30 $ ruby smtp.rb
31 digite sua senha:
32 *****
33 conexão aberta!
34 mensagem enviada!
```



Dica

Na linha 3 requisitamos o módulo `highline`, que nos permite “mascarar” a digitação da senha na linha 6.

FTP

O FTP é um protocolo para a transmissão de arquivos. Vamos requisitar um arquivo em um servidor FTP:

```
1 # encoding: utf-8
2 require "net/ftp"
3
4 host = "ftp.mozilla.org"
5 user = "anonymous"
6 pass = "eustaquiorangel@gmail.com"
7 file = "README"
8
9 begin
10     Net::FTP.open(host) do |ftp|
11         puts "Conexão FTP aberta."
12         ftp.login(user, pass)
13         puts "Requisitando arquivo ..."
14         ftp.chdir("pub")
15         ftp.get(file)
16         puts "Download efetuado."
```

```
17         puts File.read(file)
18     end
19 rescue => exception
20     puts "ERRO: #{exception}"
21 end
22
23 $ ruby ftp.rb
24 Conexão FTP aberta.
25 Requisitando arquivo ...
26 Download efetuado.
27 Welcome to ftp.mozilla.org!
28 This is a distribution point for software and developer tools related to the
29 Mozilla project. For more information, see our home page:
30 ...
```

Podemos também enviar arquivos utilizando o método `put(local, remoto)`.

POP3

Para “fechar o pacote” de e-mail, temos a classe POP3, que lida com o protocolo POP3, que é utilizado para **receber** emails. Troque o servidor, usuário e senha para os adequados no código seguinte:

```
1  # encoding: utf-8
2  require "net/pop"
3  require "highline/import"
4
5  user = "eustaquiorangel@gmail.com"
6  pass = ask("digite sua senha:") {|q| q.echo=""}
7
8  pop = Net::POP3.new("pop.gmail.com", 995)
9  pop.enable_ssl(OpenSSL::SSL::VERIFY_NONE)
10
11 begin
12     pop.start(user, pass) do |pop|
13         if pop.mails.empty?
14             puts "Sem emails!"
15             return
16         end
17         pop.each do |msg|
18             puts msg.header
19         end
20     end
21 rescue => exception
22     puts "ERRO: #{exception}"
```

```

23 end
24
25 $ ruby pop3.rb
26 digite sua senha:
27 *****
28 Return-Path: <eustaquiorangel@gmail.com>
29 Received: from localhost ([186.222.196.152])
30 by mx.google.com with ESMTPS id x15sm1427881vcs.32.2011.07.06.14.14.13
31 (version=TLSv1/SSLv3 cipher=OTHER);
32 Wed, 06 Jul 2011 14:14:17 -0700 (PDT)
33 Message-ID: <4e14d029.8f83dc0a.6a32.5cd7@mx.google.com>
34 Date: Wed, 06 Jul 2011 14:14:17 -0700 (PDT)
35 From: eustaquiorangel@gmail.com
36 Subject: Teste de SMTP no Ruby

```

HTTP

O HTTP é talvez o mais famoso dos protocolos, pois, apesar dos outros serem bastante utilizados, esse é o que dá mais as caras nos navegadores por aí, quando acessamos vários sites. É só dar uma olhada na barra de endereço do navegador que sempre vai ter um `http://` (ou `https://`, como vamos ver daqui a pouco) por lá.

Vamos utilizar o protocolo para ler o conteúdo de um site (o meu, nesse caso) e procurar alguns elementos HTML H1 (com certeza o conteúdo vai estar diferente quando você rodar isso):

```

1 require "net/http"
2
3 host = Net::HTTP.new("eustaquiorangel.com", 80)
4 resposta = host.get("/")
5 return if resposta.message != "OK"
6 puts resposta.body.scan(/<h1>.*</h1>/)
7
8 $ ruby http1.rb
9 <h1>Blog do TaQ</h1>
10 <h1><a href="/posts/dando_um_novo_g_s_para_o_plugin_snipmate_do_vim">Dando um\
11 novo gás para o plugin Snipmate do Vim</a></h1>
12 <h1>Artigos anteriores</h1>
13 <h1>Recomendados!</h1>
14 <h1>Busca</h1>
15 <h1>Twitter</h1>

```

Abrir um fluxo HTTP é muito fácil, mas dá para ficar mais fácil ainda! Vamos usar o `OpenURI`, que abre HTTP, HTTPS e FTP, o que vai nos dar resultados similares ao acima:

```

1 require "open-uri"
2
3 resposta = open("http://eustaquiorangel.com")
4 puts resposta.read.scan(/<h1>.*<\h1>/)

```

Podemos melhorar o código usando um *parser* para seleccionar os elementos. Lembrando que já utilizamos a Nokogiri para XML, podemos utilizar também para HTTP:

```

1 require "rubygems"
2 require "open-uri"
3 require "nokogiri"
4
5 doc = Nokogiri::HTML(open("http://eustaquiorangel.com"))
6 puts doc.search("h1").map {|elemento| elemento.text}
7
8 $ ruby http3.rb
9 Blog do TaQ
10 Dando um novo gás para o plugin Snipmate do Vim
11 Artigos anteriores
12 Recomendados!
13 Busca
14 Twitter

```

Aproveitando que estamos falando de HTTP, vamos ver como disparar um servidor web, o WEBrick, que já vem com Ruby:

```

1 require "webrick"
2 include WEBrick
3
4 s = HTTPServer.new(:Port=>2000, :DocumentRoot=>Dir.pwd)
5 trap("INT") { s.shutdown }
6 s.start
7
8 $ ruby webrick.rb
9 [2011-07-06 20:56:54] INFO WEBrick 1.3.1
10 [2011-07-06 20:56:54] INFO ruby 1.9.2 (2010-08-18) [i686-linux]
11 [2011-07-06 20:56:54] WARN TCPServer Error: Address already in use - bind(2)
12 [2011-07-06 20:56:54] INFO WEBrick::HTTPServer#start: pid=19677 port=2000

```

HTTPS

O HTTPS é o primo mais seguro do HTTP. Sempre o utilizamos quando precisamos de uma conexão segura onde podem ser enviados dados sigilosos como senhas, dados de cartões de crédito e coisas do tipo que, se caírem nas mãos de uma turma por aí que gosta de fazer coisas erradas, vai nos dar algumas belas dores de cabeça depois.

Podemos acessar HTTPS facilmente:

```
1 require "net/https"
2 require "highline/import"
3
4 user = "user"
5 pass = ask("digite sua senha") {|q| q.echo=""}
6
7 begin
8   site = Net::HTTP.new("api.del.icio.us",443)
9   site.use_ssl = true
10  site.start do |http|
11    req = Net::HTTP::Get.new('/v1/tags/get')
12    req.basic_auth(user,pass)
13    response = http.request(req)
14    print response.body
15  end
16 rescue => exception
17   puts "erro: #{e}"
18 end
```

SSH

O SSH é ao mesmo tempo um programa e um protocolo, que podemos utilizar para estabelecer conexões seguras e criptografadas com outro computador. É um telnet super-vitaminado, com várias vantagens que só eram desconhecidas (e devem continuar) por um gerente de uma grande empresa que prestei serviço, que acreditava que o bom mesmo era telnet ou FTP, e SSH era ... “inseguro”. Sério! O duro que esse tipo de coisa, infelizmente, é comum entre pessoas em cargo de liderança em tecnologia por aí, e dá para arrumar umas boas discussões inúteis por causa disso. Mas essa é outra história ...

Vamos começar a trabalhar com o SSH e abrir uma conexão e executar alguns comandos. Para isso precisamos da *gem* net-ssh:

```
1 gem install net-ssh
```

E agora vamos rodar um programa similar ao seguinte, onde você deve alterar o host, usuário e senha para algum que você tenha acesso:

```

1  # encoding: utf-8
2  require "rubygems"
3  require "net/ssh"
4  require "highline/import"
5
6  host = "eustaquiorangel.com"
7  user = "taq"
8  pass = ask("digite sua senha") {|q| q.echo="" }
9
10 begin
11   Net::SSH.start(host,user,:password=>pass) do |session|
12     puts "Sessão SSH aberta!"
13     session.open_channel do |channel|
14       puts "Canal aberto!"
15       channel.on_data do |ch,data|
16         puts "> #{data}"
17       end
18       puts "Executando comando ..."
19       channel.exec "ls -lah"
20     end
21     session.loop
22   end
23 rescue => exception
24   puts "ERRO:#{exception}"
25   puts exception.backtrace
26 end
27
28 $ ruby code/ssh.rb
29 digite sua senha
30 *****
31 Sessão SSH aberta!
32 Canal aberto!
33 Executando comando
34 > total 103M
35 drwxr-xr-x 6 taq taq 4.0K Jun 17 19:10
36 ...

```

XML-RPC

XML-RPC²⁹ é, segundo a descrição em seu site:

É uma especificação e um conjunto de implementações que permitem á softwares rodando em sistemas operacionais diferentes, rodando em diferentes ambientes, fazerem chamadas de procedures pela internet.

²⁹<http://www.xmlrpc.com>

A chamada de *procedures* remotas é feita usando HTTP como transporte e XML como o *encoding*. XML-RPC é desenhada para ser o mais simples possível, permitindo estruturas de dados completas serem transmitidas, processadas e retornadas.

Tentando dar uma resumida, você pode escrever métodos em várias linguagens rodando em vários sistemas operacionais e acessar esses métodos através de várias linguagens e vários sistemas operacionais.

Antes de mais nada, vamos criar um servidor que vai responder as nossas requisições, fazendo algumas operações matemáticas básicas, que serão **adição** e **divisão**:

```

1  require "xmlrpc/server"
2
3  server = XMLRPC::Server.new(8081)
4
5  # somando números
6  server.add_handler("soma") do |n1,n2|
7    {"resultado"=>n1+n2}
8  end
9
10 # dividindo e retornando o resto
11 server.add_handler("divide") do |n1,n2|
12   {"resultado"=>n1/n2, "resto"=>n1%n2}
13 end
14 server.serve
15
16 $ ruby rpcserver.rb
17 [2011-07-06 21:16:07] INFO WEBrick 1.3.1
18 [2011-07-06 21:16:07] INFO ruby 1.9.2 (2010-08-18) [i686-linux]
19 [2011-07-06 21:16:07] INFO WEBrick::HTTPServer#start: pid=20414 port=8081

```

Agora vamos fazer um cliente para testar (você pode usar qualquer outra linguagem que suporte RPC que desejar):

```

1  # encoding: utf-8
2  require "xmlrpc/client"
3
4  begin
5    client = XMLRPC::Client.new("localhost", "/RPC2", 8081)
6    resp = client.call("soma", 5, 3)
7    puts "O resultado da soma é #{resp['resultado']}"
8
9    resp = client.call("divide", 11, 4)
10   puts "O resultado da divisao é #{resp['resultado']} e o resto é #{resp['r\
11 esto']}"
12 rescue => exception
13   puts "ERRO: #{exception}"

```

```
14 end
15
16 $ ruby rpcclient.rb
17 O resultado da soma é 8
18 O resultado da divisao é 2 e o resto é 3
```

Vamos acessar agora o servidor de outras linguagens.

Python

```
1 # coding: utf-8
2 import xmlrpclib
3
4 server = xmlrpclib.Server("http://localhost:8081")
5 result = server.soma(5,3)
6 print "O resultado da soma é:",result["resultado"]
7
8 result = server.divide(11,4)
9 print "O resultado da divisão é",result["resultado"],"e o resto é",result["re\
10 sto"]
11
12 $ python rpcclient.py
13 O resultado da soma é: 8
14 O resultado da divisão é 2 e o resto é 3
```

PHP

```
1 <?php
2 // soma
3 $request = xmlrpc_encode_request("soma", array(5,3));
4 $context = stream_context_create(array('http' => array('method' => "POST"\
5 , 'header' => "Content-Type: text/xml", 'content' => $request)));
6
7 $file = file_get_contents("http://localhost:8081", false, $context);
8 $response = xmlrpc_decode($file);
9 if ($response && xmlrpc_is_fault($response)) {
10     trigger_error("xmlrpc: $response[faultString] ($response[faultCode])"\
11 );
12 } else {
13     print "O resultado da soma é ".$response["resultado"]."\n";
14 }
15
16 // divisão
17 $request = xmlrpc_encode_request("divide", array(11,4));
18 $context = stream_context_create(array('http' => array('method' => "POST"\
```

```

19 , 'header' => "Content-Type: text/xml", 'content' => $request));
20
21     $file = file_get_contents("http://localhost:8081", false, $context);
22     $response = xmlrpc_decode($file);
23     if ($response && xmlrpc_is_fault($response)) {
24         trigger_error("xmlrpc: $response[faultString] ($response[faultCode])" \
25     );
26     } else {
27         print "O resultado da divisão é ".$response["resultado"]." e o resto \
28     é ".$response["resto"]."\n";
29     }
30 ?>
31
32 $ php rpcclient.php
33 O resultado da soma é 8
34 O resultado da divisão é 2 e o resto é 3

```

Java

Em Java vamos precisar do Apache XML-RPC³⁰:

```

1  import java.net.URL;
2  import java.util.Vector;
3  import java.util.HashMap;
4  import org.apache.xmlrpc.common.*;
5  import org.apache.xmlrpc.client.*;
6
7  public class RPCCClient {
8      public static void main(String args[]){
9          try{
10             Vector <Integer>params;
11             XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
12             config.setServerURL(new URL("http://localhost:8081/RPC2"));
13             XmlRpcClient server = new XmlRpcClient();
14             server.setConfig(config);
15
16             params = new Vector<Integer>();
17             params.addElement(new Integer(5));
18             params.addElement(new Integer(3));
19
20             HashMap result = (HashMap) server.execute("soma",params);
21             int sum = ((Integer) result.get("resultado")).intValue();
22             System.out.println("O resultado da soma é "+Integer.toString(sum)\
23     );

```

³⁰<http://ws.apache.org/xmlrpc>

```
24
25     params = new Vector<Integer>();
26     params.addElement(new Integer(11));
27     params.addElement(new Integer(4));
28     result = (HashMap) server.execute("divide",params);
29
30     int divide = ((Integer) result.get("resultado")).intValue();
31     int resto = ((Integer) result.get("resto")).intValue();
32     System.out.println("O resultado da divisão é "+Integer.toString(s\
33 um)+" e o resto é: "+Integer.toString(resto));
34     }catch(Exception error){
35         System.err.println("erro:"+error.getMessage());
36     }
37 }
38 }
39
40 $ javac -classpath commons-logging-1.1.jar:ws-commons-util-1.0.2.jar:xmlrpc-cl\
41 ient-3.1.3.jar:xmlrpc-common-3.1.3.jar: RPCClient.java
42 $ java -classpath commons-logging-1.1.jar:ws-commons-util-1.0.2.jar:xmlrpc-cl\
43 ient-3.1.3.jar:xmlrpc-common-3.1.3.jar: RPCClient
44 O resultado da soma é 8
45 O resultado da divisão é 8 e o resto é: 3
```

JRuby

Vamos instalar JRuby para dar uma olhada em como integrar Ruby com Java, usando a RVM. Antes de mais nada, pedimos para ver as notas da RVM e procurar as instruções para instalar JRuby:

```
1 $ rvm requirements
2 # For jruby:
3 sudo apt-get --no-install-recommends install g++ openjdk-7-jre-headless
4
5 $ rvm install jruby
6 $ rvm use jruby
7 $ jruby -v
8 $ jruby 1.7.2 (1.9.3p327) 2013-01-04 302c706 on OpenJDK Server VM 1.6.0_27-b2\
9 7 [linux-i386]
```

Precisamos inserir as classes do JRuby no CLASSPATH do Java. Teste as duas opções abaixo, se você estiver em um SO que suporte o comando `locate`, a primeira é bem mais rápida, do contrário, use a segunda.

Primeiro utilizando `locate`:

```
1 $ export CLASSPATH=$CLASSPATH:$(locate jruby.jar)::
```

Se o comando `locate` não for encontrado/suportado, utilize `find`:

```
1 $ export CLASSPATH=$CLASSPATH:$(find ~ -iname 'jruby.jar')::
```



Desafio 7

Tentem entender como que eu adicionei as classes necessárias para o JRuby no CLASSPATH do Java ali acima.

Agora fazendo um pequeno programa em Ruby:

```
1 puts "digite seu nome:"
2 nome = gets.chomp
3 puts "oi, #{nome}!"
4
5 $ jruby jruby.rb
6 $ java jruby
7 digite seu nome:
8 taq
9 oi, taq!
```

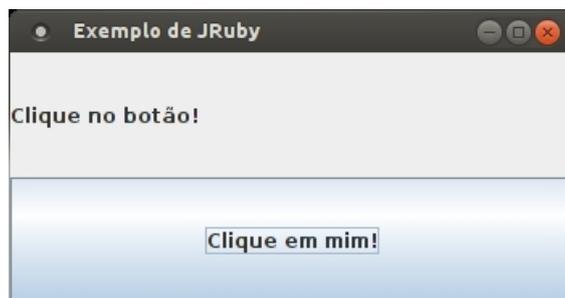
Utilizando classes do Java de dentro do Ruby

Vamos criar um programa chamado `gui.rb`:

```
1 # encoding: utf-8
2 require "java"
3
4 %w(JFrame JLabel JPanel JButton).each { |c| java_import("javax.swing.#{c}") }
5
6 class Alistener
7   include java.awt.event.ActionListener
8   def actionPerformed(event)
9     puts "Botão clicado!"
10  end
11 end
12 listener = Alistener.new
13
14 frame = JFrame.new
15 label = JLabel.new("Clique no botão!")
16 panel = JPanel.new
17
18 button = JButton.new("Clique em mim!")
19 button.addActionListener(listener)
20
21 panel.setLayout(java.awt.GridLayout.new(2,1))
22 panel.add(label)
23 panel.add(button)
24
25 frame.setTitle("Exemplo de JRuby")
26 frame.getContentPane().add(panel)
27 frame.pack
28 frame.defaultCloseOperation = JFrame::EXIT_ON_CLOSE
29 frame.setVisible(true)
```

Compilando e rodando o programa:

```
1 $ jrubbyc gui.rb
2 $ java gui
```



Usando GUI do Java em Ruby

Pudemos ver que criamos a classe `Alistener` com a interface, no caso aqui com um comportamento de módulo, `java.awt.event.ActionListener`, ou seja, JRuby nos permite utilizar interfaces do Java como se fossem módulos de Ruby! E tem mais, podemos fazer com que nossas classes em Ruby herdem de classes do Java, primeiro, escrevendo o arquivo `Carro.java`:

```
1 // Carro.java
2 public class Carro {
3     private String marca, cor, modelo;
4     private int tanque;
5
6     public Carro(String marca, String cor, String modelo, int tanque) {
7         this.marca = marca;
8         this.cor = cor;
9         this.modelo = modelo;
10        this.tanque = tanque;
11    }
12
13    public String toString() {
14        return "Marca: "+this.marca + "\n" + "Cor: "+this.cor + "\n" + "Model\
15o:"+this.modelo + "\n" + "Tanque:"+this.tanque;
16    }
17 }
```

e agora o arquivo `carro_java.rb`:

```
1 # carro.rb
2 require "java"
3 java_import("Carro")
4
5 carro = Carro.new("VW", "prata", "polo", 40)
6 puts carro
7
8 class Mach5 < Carro
9   attr_reader :tanque_oxigenio
10
11   def initialize(marca, cor, modelo, tanque, tanque_oxigenio)
12     super(marca, cor, modelo, tanque)
13     @tanque_oxigenio = tanque_oxigenio
14   end
15
16   def to_s
17     "#{super}\nTanque oxigenio: #{@tanque_oxigenio}"
18   end
19 end
20
21 puts "*" * 25
22 mach5 = Mach5.new("PopsRacer", "branco", "Mach5", 50, 10)
23 puts mach5
24
25 $ javac Carro.java
26 $ jrubyc carro_java.rb
27 $ java carro_java
28 Marca: VW
29 Cor: prata
30 Modelo: polo
31 Tanque: 40
32 *****
33 Marca: PopsRacer
34 Cor: branco
35 Modelo: Mach5
36 Tanque: 50
37 Tanque oxigenio: 10
```

Usando classes do Ruby dentro do Java

Existe um jeito de fazer isso, mas vão por mim: não compensa pois vocês vão xingar muito o Java. Para maiores referências, podem consultar o site oficial de scripting para Java em <http://java.net/projects/scripting/>³¹.

³¹<http://java.net/projects/scripting/>

Banco de dados

Vamos utilizar uma interface uniforme para acesso aos mais diversos bancos de dados suportados em Ruby através da interface Sequel[^sequel]. Para instalá-la, é só utilizar a *gem* sequel: [^sequel]: <http://sequel.rubyforge.org/>

```
1 gem install sequel
```

Abrindo a conexão

Vamos abrir e fechar a conexão com o banco:

```
1 con = Sequel.mysql(:user=>"root", :password=>"aluno", :host=>"localhost", :datab\
2 ase=>"aluno")
3 => #<Sequel::MySQL::Database: "mysql://root:aluno@localhost/aluno">
```

Para dar uma encurtada no código e praticidade maior, vamos usar um bloco logo após conectar, para onde vai ser enviado o *handle* da conexão:

```
1 require "sequel"
2
3 Sequel.mysql(:user=>"aluno", :password=>"aluno", :host=>"localhost", :database=>\
4 "alunos") do |con|
5   p con
6 end
```

Desse modo sempre que a conexão for aberta, ela será automaticamente fechada no fim do bloco.



Dica Para trocar o banco de dados, podemos alterar apenas o método de conexão. Se, por exemplo, quisermos utilizar o SQLite3, podemos utilizar:

```
1 require "sequel"
2 Sequel.sqlite("alunos.sqlite3") do |con|
3   p con
4 end
5
6 $ ruby db2.rb
7 #<Sequel::SQLite::Database: "sqlite:/alunos.sqlite3">
```

Consultas que não retornam dados

Vamos criar uma tabela nova para usamos no curso, chamada `alunos` e inserir alguns valores:

```
1 # encoding: utf-8
2 require "sequel"
3
4 Sequel.mysql(:user=>"aluno",:password=>"aluno",:host=>"localhost",:database=>\
5 "alunos") do |con|
6   con.run("drop table if exists alunos")
7   sql = <<FIM
8   create table alunos (
9     id int(6) not null,
10    nome varchar(50) not null)
11   FIM
12   con.run(sql)
13
14   con[:alunos].insert(:id=>1,:nome=>'João')
15   con[:alunos].insert(:id=>2,:nome=>'José')
16   con[:alunos].insert(:id=>3,:nome=>'Antonio')
17   con[:alunos].insert(:id=>4,:nome=>'Maria')
18 end
19
20 $ ruby db3.rb
21 mysql -h localhost -u root -p aluno
22 Enter password:
23 mysql> select * from alunos;
24 +----+-----+
25 | id | nome   |
26 +----+-----+
27 | 1 | João   |
28 | 2 | José   |
29 | 3 | Antonio |
30 | 4 | Maria  |
31 +----+-----+
32 4 rows in set (0.03 sec)
```

Atualizando um registro

```
1 # encoding: utf-8
2 require "sequel"
3
4 Sequel.mysql(:user=>"aluno",:password=>"aluno",:host=>"localhost",:database=>\
5 "alunos") do |con|
6   puts con[:alunos].where(:id=>4).update(:nome=>"Mário")
7 end
8
9 $ ruby db13.rb
```

Apagando um registro

```
1 # encoding: utf-8
2 require "sequel"
3
4 Sequel.mysql(:user=>"aluno", :password=>"aluno", :host=>"localhost", :database=>\
5 "alunos") do |con|
6   con[:alunos].insert(:id=>5, :nome=>"Teste")
7   puts con[:alunos].where(:id=>5).delete
8 end
9
10 $ ruby db14.rb
```

Consultas que retornam dados

Vamos recuperar alguns dados do nosso banco, afinal, essa é a operação mais costumeira, certo? Para isso, vamos ver duas maneiras. Primeiro, da maneira “convencional”:

```
1 # encoding: utf-8
2 require "sequel"
3
4 Sequel.mysql(:user=>"aluno", :password=>"aluno", :host=>"localhost", :database=>\
5 "alunos") do |con|
6   con[:alunos].each do |row|
7     puts "id: #{row[:id]} nome: #{row[:nome]}"
8   end
9 end
10
11 $ ruby db4.rb
12 id: 1 nome: João
13 id: 2 nome: José
14 id: 3 nome: Antonio
15 id: 4 nome: Mário
```

Podemos recuperar todos as linhas de dados de uma vez usando `all`:

```

1 require "sequel"
2
3 Sequel.mysql(:user=>"aluno", :password=>"aluno", :host=>"localhost", :database=>\
4 "alunos") do |con|
5   rows = con[:alunos].all
6   puts "#{rows.size} registros recuperados"
7   rows.each {|row| puts "id: #{row[:id]} nome: #{row[:nome]}"}
8 end
9
10 $ ruby db5.rb
11 4 registros recuperados
12 id: 1 nome: João
13 id: 2 nome: José
14 id: 3 nome: Antonio
15 id: 4 nome: Mário

```

Ou se quisermos somente o primeiro registro:

```

1 require "sequel"
2
3 Sequel.mysql(:user=>"aluno", :password=>"aluno", :host=>"localhost", :database=>\
4 "alunos") do |con|
5   row = con[:alunos].first
6   puts "id: #{row[:id]} nome: #{row[:nome]}"
7 end
8
9 $ ruby db8.rb
10 id: 1 nome: João

```

Comandos preparados

Agora vamos consultar registro por registro usando comandos preparados com argumentos variáveis, o que vai nos dar resultados similares mas muito mais velocidade quando executando a mesma consulta SQL trocando apenas os argumentos que variam:

```

1 require "sequel"
2
3 Sequel.mysql(:user=>"aluno", :password=>"aluno", :host=>"localhost", :database=>\
4 "alunos") do |con|
5   ds = con[:alunos].filter(:id=>:$i)
6   ps = ds.prepare(:select, :select_by_id)
7   (1..4).each do |id|
8     print "procurando id #{id} ... "
9     row = ps.call(:i=>id)

```

```
10     puts "#{row.first[:nome]}"
11   end
12 end
13
14 $ ruby db9.rb
15 procurando id 1 ... João
16 procurando id 2 ... José
17 procurando id 3 ... Antonio
18 procurando id 4 ... Mário
```

Metadados

Vamos dar uma examinada nos dados que recebemos de nossa consulta e na estrutura de uma tabela:

```
1 require "sequel"
2
3 Sequel.mysql(:user=>"aluno", :password=>"aluno", :host=>"localhost", :database=>\
4 "alunos") do |con|
5   p con[:alunos].columns
6   p con.schema(:alunos)
7 end
8
9 $ ruby db10.rb
10 [{:mysql_flags=>36865, :type_name=>"INTEGER",
11 :dbi_type=>DBI::Type::Integer, :unique=>false, :mysql_length=>6,
12 :mysql_type=>3, :mysql_max_length=>1, :precision=>6,
13 :indexed=>false, :sql_type=>4, :mysql_type_name=>"INT", :scale=>0,
14 :name=>"id", :primary=>false, :nullable=>false},
15 {:mysql_flags=>4097, :type_name=>"VARCHAR",
16 :dbi_type=>DBI::Type::Varchar, :unique=>false, :mysql_length=>50,
17 :mysql_type=>253, :mysql_max_length=>7, :precision=>50,
18 :indexed=>false, :sql_type=>12, :mysql_type_name=>"VARCHAR",
19 :scale=>0, :name=>"nome", :primary=>false, :nullable=>false}]
```

ActiveRecord

Agora vamos ver uma forma de mostrar que é possível utilizar o “motorzão” ORM do Rails sem o Rails, vamos ver como criar e usar um modelo da nossa tabela `alunos`, antes atendendo à uma pequena requisição do ActiveRecord, que pede uma coluna chamada `id` como chave primária:

```

1  mysql -h localhost -u root -p aluno
2  Enter password:
3  mysql> alter table alunos add primary key (id);
4  Query OK, 4 rows affected (0.18 sec)
5  Records: 4 Duplicates: 0 Warnings: 0
6  mysql> desc alunos;
7  +-----+-----+-----+-----+-----+-----+
8  | Field | Type          | Null | Key | Default | Extra |
9  +-----+-----+-----+-----+-----+-----+
10 | id    | int(6)        | NO   | PRI | NULL    |       |
11 | nome  | varchar(50)   | NO   |     | NULL    |       |
12 +-----+-----+-----+-----+-----+
13 2 rows in set (0.00 sec)
14 mysql> quit
15 Bye

```

Agora, nosso programa:

```

1  # encoding: utf-8
2  require "rubygems"
3  require "active_record"
4
5  # estabelecendo a conexão
6  ActiveRecord::Base.establish_connection({
7    :adapter => "mysql",
8    :database => "aluno",
9    :username => "root",
10   :password => "aluno"
11 })
12
13 # criando o mapeamento da classe com a tabela
14 # (espera aí é só isso???)
15 class Aluno < ActiveRecord::Base
16 end
17
18 # pegando a coleção e usando o seu iterador
19 for aluno in Aluno.all
20   puts "id: #{aluno.id} nome: #{aluno.nome}"
21 end
22
23 # atualizando o nome de um aluno
24 aluno = Aluno.find(3)
25 puts "encontrei #{aluno.nome}"
26 aluno.nome = "Danilo"
27 aluno.save

```

Rodando o programa:

```
1 $ ruby arec.rb
2 id: 1 nome: João
3 id: 2 nome: José
4 id: 3 nome: Antonio
5 id: 4 nome: Maria
6 encontrei Antonio
```

Se rodarmos novamente, vamos verificar que o registro foi alterado, quando rodamos o programa anteriormente:

```
1 $ ruby arec.rb
2 id: 1 nome: João
3 id: 2 nome: José
4 id: 3 nome: Danilo
5 id: 4 nome: Maria
6 encontrei Danilo
```

Escrevendo extensões para Ruby, em C

Se quisermos incrementar um pouco a linguagem usando linguagem C para

- Maior velocidade
- Recursos específicos do sistema operacional que não estejam disponíveis na implementação padrão
- Algum desejo mórbido de lidar com segfaults e ponteiros nulos
- Todas as anteriores

podemos escrever facilmente extensões em C.

Vamos criar um módulo novo chamado `Curso` com uma classe chamada `Horario` dentro dele, que vai nos permitir cadastrar uma descrição da instância do objeto no momento em que o criarmos, e vai retornar a data e a hora correntes em dois métodos distintos.

Que uso prático isso teria não sei, mas vamos relevar isso em função do exemplo didático do código apresentado. ;-)

A primeira coisa que temos que fazer é criar um arquivo chamado `extconf.rb`, que vai usar o módulo `mkmf` para criar um `Makefile` que irá compilar os arquivos da nossa extensão:

```
1 require "mkmf"
2 extension_name = "curso"
3 dir_config(extension_name)
4 create_makefile(extension_name)
```

Vamos assumir essa sequência de código como a nossa base para fazer extensões, somente trocando o nome da extensão na variável `extension_name`.

Agora vamos escrever o fonte em C da nossa extensão, como diria Jack, O Estripador, “por partes”. Crie um arquivo chamado `curso.c` com o seguinte conteúdo:

```
1 #include <ruby.h>
2 #include <time.h>
3
4 VALUE modulo, classe;
5
6 void Init_curso(){
7     modulo = rb_define_module("Curso");
8     classe = rb_define_class_under(modulo, "Horario", rb_cObject);
9 }
```

Opa! Já temos algumas coisas definidas ali! Agora temos que criar um Makefile³² para compilarmos nossa extensão. O bom que ele é gerado automaticamente a partir do nosso arquivo `extconf.rb`:

```
1 $ ruby extconf.rb
2 creating Makefile
```

E agora vamos executar o `make` para ver o que acontece:

```
1 $ make
2 compiling curso.c
3 linking shared-object curso.so
```

Dando uma olhada no diretório, temos:

```
1 $ ls *.so
2 curso.so
```

Foi gerado um arquivo `.so`, que é um arquivo de bibliotecas compartilhadas do GNU/Linux (a analogia no mundo Windows é uma DLL) com o nome que definimos para a extensão, com a extensão apropriada. Vamos fazer um teste no `irb` para ver se tudo correu bem:

```
1 require "./curso"
2 => true
3 horario = Curso::Horario.new
4 => #<Curso::Horario:0x991aa4c>
```

Legal, já temos nosso primeiro módulo e classe vindos diretamente do C! Vamos criar agora o método construtor, alterando nosso código fonte C:

³²http://pt.wikibooks.org/wiki/Programar_em_C/Makefiles

```

1  #include <ruby.h>
2  #include <time.h>
3
4  VALUE modulo, classe;
5
6  VALUE t_init(VALUE self, VALUE valor){
7      rb_iv_set(self, "@descricao", valor);
8      return self;
9  }
10
11 void Init_curso(){
12     modulo = rb_define_module("Curso");
13     classe = rb_define_class_under(modulo, "Horario", rb_cObject);
14     rb_define_method(classe, "initialize", t_init, 1);
15 }

```

Vamos testar, lembrando de rodar o make para compilar novamente o código:

```

1  require "./curso"
2  => true
3  horario = Curso::Horario.new
4  ArgumentError: wrong number of arguments(0 for 1)
5  from (irb):2:in 'initialize'
6  from (irb):2:in 'new'
7  from (irb):2
8  from /home/aluno/.rvm/rubies/ruby-1.9.2-p180/bin/irb:16:in '<main>'
9  horario = Curso::Horario.new(:teste)
10 => #<Curso::Horario:0x8b9e5e4 @descricao=:teste>

```

Foi feita uma tentativa de criar um objeto novo sem passar argumento algum no construtor, mas ele estava esperando um parâmetro, definido com o número 1 no final de `rb_define_method`.

Logo após criamos o objeto enviando um `Symbol` e tudo correu bem, já temos o nosso construtor!

Reparem como utilizamos `rb_iv_set` (algo como Ruby Instance Variable Set) para criar uma variável de instância com o argumento enviado. Mas a variável de instância continua sem um método para ler o seu valor, presa no objeto:

```

1  horario.descricao
2  NoMethodError: undefined method 'descricao' for
3  #<Curso::Horario:0x8b9e5e4 @descricao=:teste>
4  from (irb):4

```

Vamos criar um método para acessá-la:

```

1  #include <ruby.h>
2  #include <time.h>
3
4  VALUE modulo, classe;
5
6  VALUE t_init(VALUE self, VALUE valor){
7      rb_iv_set(self, "@descricao", valor);
8      return self;
9  }
10
11 VALUE descricao(VALUE self){
12     return rb_iv_get(self, "@descricao");
13 }
14
15 void Init_curso(){
16     modulo = rb_define_module("Curso");
17     classe = rb_define_class_under(modulo, "Horario", rb_cObject);
18     rb_define_method(classe, "initialize", t_init, 1);
19     rb_define_method(classe, "descricao", descricao, 0);
20 }

```

Rodando novamente:

```

1  require "./curso"
2  => true
3  horario = Curso::Horario.new(:teste)
4  => #<Curso::Horario:0x8410d04 @descricao=:teste>
5  horario.descricao
6  => :teste

```

Agora para fazer uma graça vamos definir dois métodos que retornam a data e a hora corrente, como Strings. A parte mais complicada é pegar e formatar isso em C. Convém prestar atenção no modo que é alocada uma String nova usando `rb_str_new2`.



Dica

Apesar dos nomes parecidos, `rb_str_new` espera dois argumentos, uma String e o comprimento, enquanto `rb_str_new2` espera somente uma String terminada com nulo e é bem mais prática na maior parte dos casos.

```
1  #include <ruby.h>
2  #include <time.h>
3
4  VALUE modulo, classe;
5
6  VALUE t_init(VALUE self, VALUE valor){
7      rb_iv_set(self, "@descricao", valor);
8      return self;
9  }
10
11 VALUE descricao(VALUE self){
12     return rb_iv_get(self, "@descricao");
13 }
14
15 struct tm *get_date_time() {
16     time_t dt;
17     struct tm *dc;
18     time(&dt);
19     dc = localtime(&dt);
20     return dc;
21 }
22
23 VALUE data(VALUE self){
24     char str[15];
25     struct tm *dc = get_date_time();
26     sprintf(str, "%02d/%02d/%04d", dc->tm_mday, dc->tm_mon+1, dc->tm_year+1900);
27     return rb_str_new2(str);
28 }
29
30 VALUE hora(VALUE self){
31     char str[15];
32     struct tm *dc = get_date_time();
33     sprintf(str, "%02d:%02d:%02d", dc->tm_hour, dc->tm_min, dc->tm_sec);
34     return rb_str_new2(str);
35 }
36
37 void Init_curso(){
38     modulo = rb_define_module("Curso");
39     classe = rb_define_class_under(modulo, "Horario", rb_cObject);
40     rb_define_method(classe, "initialize", t_init, 1);
41     rb_define_method(classe, "descricao", descricao, 0);
42     rb_define_method(classe, "data", data, 0);
43     rb_define_method(classe, "hora", hora, 0);
44 }
```

Rodando o programa:

```
1 horario = Curso::Horario.new(:teste)
2 => #<Curso::Horario:0x896b6dc @descricao=:teste>
3 horario.descricao
4 => :teste
5 horario.data
6 => "14/07/2011"
7 horario.hora
8 => "15:33:27"
```

Tudo funcionando perfeitamente! Para maiores informações de como criar extensões para Ruby, uma boa fonte de consultas é http://www.rubycentral.com/pickaxe/ext_ruby.html³³.

³³http://www.rubycentral.com/pickaxe/ext_ruby.html

Garbage collector

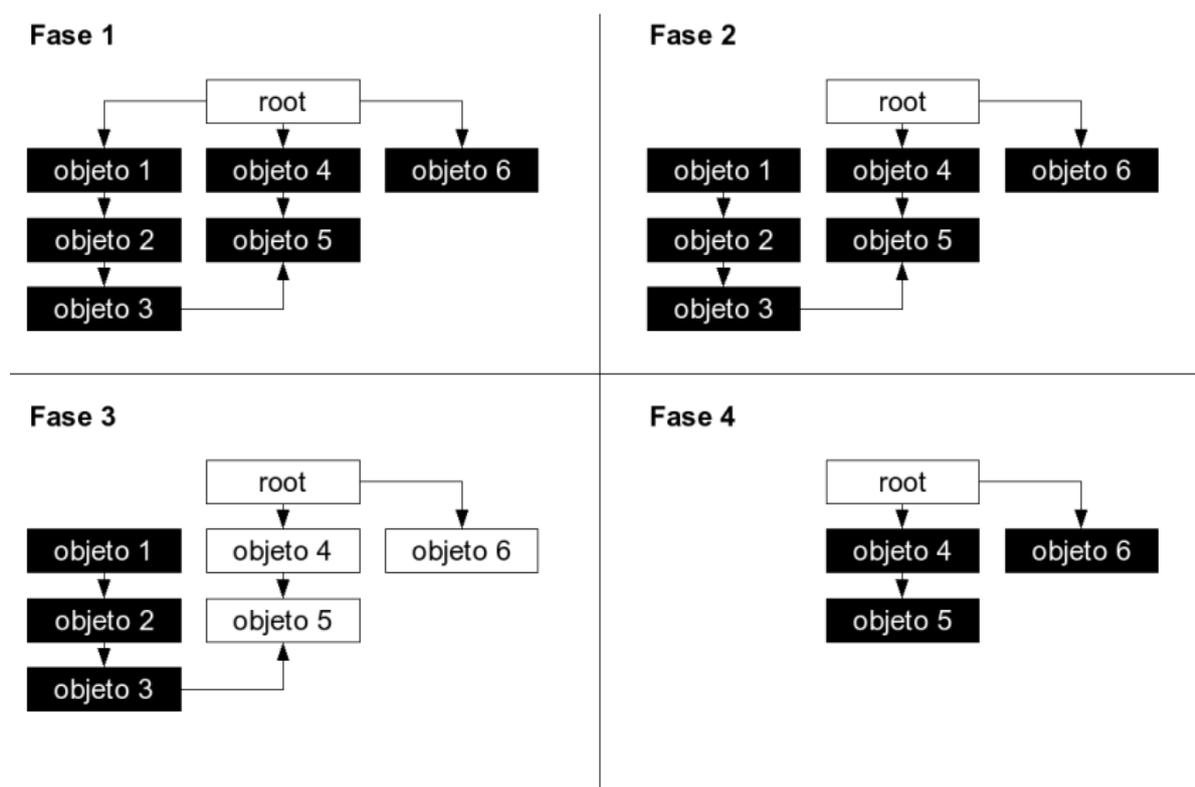
Vamos aproveitar que estamos falando de coisa de um nível mais baixo (não, não é de política) e vamos investigar como funciona o garbage collector do Ruby. Várias linguagens modernas tem um *garbage collector*, que é quem recolhe objetos desnecessários e limpa a memória para nós. Isso evita que precisemos alocar memória sempre que criar um objeto e libera-lá após a sua utilização. Quem programa em C conhece bem `malloc` e `free`, não é mesmo? E ainda mais os famigerados *null pointer assignments*.

Em Ruby, o *garbage collector* é do tipo *mark-and-sweep*, que atua em fases separadas onde marca os objetos que não são mais necessários e depois os limpa. Vamos ver fazendo um teste prático de criar alguns objetos, invalidar algum, chamar o *garbage collector* e verificar os objetos novamente:

```
1 class Teste
2 end
3
4 t1 = Teste.new
5 t2 = Teste.new
6 t3 = Teste.new
7
8 count = ObjectSpace.each_object(Teste) do |object|
9   puts object
10 end
11 puts "#{count} objetos encontrados."
12
13 t2 = nil
```

GC.start

```
1 count = ObjectSpace.each_object(Teste) do |object|
2   puts object
3 end
4 puts "#{count} objetos encontrados."
5
6 $ ruby gc1.rb
7 #<Teste:0x850d1a8>
8 #<Teste:0x850d1bc>
9 #<Teste:0x850d1d0>
10 3 objetos encontrados.
11 #<Teste:0x850d1a8>
12 #<Teste:0x850d1d0>
13 2 objetos encontrados.
```



Garbage Collector

- Na Fase 1, todos os objetos não estão marcados como acessíveis.
- Na Fase 2, continuam do mesmo jeito, porém o *objeto 1* agora não está disponível no *root*.
- Na Fase 3, o algoritmo foi acionado, parando o programa e marcando (*mark*) os objetos que estão acessíveis.
- Na Fase 4 foi executada a limpeza (*sweep*) dos objetos não-acessíveis, e retirado o *flag* dos que estavam acessíveis (deixando-os em preto novamente), forçando a sua verificação na próxima vez que o *garbage collector* rodar.

Isso não é um livro de C mas ...

Não custa ver como uma linguagem com alocação e limpeza automática de memória quebra nosso galho. Considerem esse código:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main() {
6     char *str;
7     str = malloc(sizeof(char)*15);
8     strcpy(str, "hello world");
9     printf("%s\n", str);
10    free(str);
11    return 0;
12 }
```

Vamos compilá-lo (você tem o GCC aí, não tem?) e executá-lo:

```
1 $ gcc -o null null.c
2 $ ./null
3 hello world
```

Até aqui tudo bem. Mas agora comentem a linha 7, onde é executada malloc:

```
1 $ gcc -o null null.c
2 $ ./null
3 hello world
4 *** Error in `./null': free(): invalid pointer: 0xb7758000 ***
5 ===== Backtrace: =====
6 ...
```

Oh-oh. Como não houve alocação de memória, a chamada a free disparou uma mensagem de erro. Comentando a linha 10, onde se encontra free:

```
1 $ gcc -o null null.c
2 $ ./null
3 hello world
```

Aparentemente sem problemas, não é mesmo? Só que copiar uma String para um ponteiro de memória não inicializado pode nos dar algumas dores de cabeça ...

Isso ainda não é um livro de C, mas ...

Mas temos que aprender a verificar se um simples programa como esse tem alguma falha. Para isso, podemos utilizar o Valgrind³⁴, que é uma ferramenta ótima para esse tipo de coisa. Vamos executar o comando valgrind pedindo para verificar *memory leaks* no nosso pequeno programa, no estado em que está:

³⁴<http://valgrind.org>

```

1 $ valgrind --tool=memcheck --leak-check=yes -q ./null
2 ==8119== Use of uninitialised value of size 4
3 ==8119==    at 0x8048429: main (in /home/taq/code/ruby/conhecendo-ruby/null)
4 ==8119==
5 ...

```

Não vamos entrar a fundo no uso do Valgrind, mas isso significa que nosso programa tem um problema. Vamos tentar remover o comentário da linha 10, onde está `free`, compilar e rodar o comando `valgrind` novamente:

```

1 $ gcc -o null null.c
2 $ valgrind --tool=memcheck --leak-check=yes -q ./null
3 ==8793== Use of uninitialised value of size 4
4 ==8793==    at 0x8048459: main (in /home/taq/code/ruby/conhecendo-ruby/null)
5 ==8793==

```

Ainda não deu certo, e vamos voltar no comportamento já visto de erro do programa na hora em que executarmos ele. Vamos remover agora o comentário da linha 7, onde está `malloc`, e rodar novamente o `valgrind`:

```

1 $ gcc -o null null.c
2 $ valgrind --tool=memcheck --leak-check=yes -q ./null
3 hello world

```

Agora temos certeza de que está tudo ok! O Valgrind é uma ferramenta muito poderosa que quebra altos galhos.



Dica

Para termos um retorno exato do Valgrind de onde está o nosso problema, compilem o programa utilizando a opção `-g`, que vai inserir informações de *debugging* no executável. Se comentarmos novamente a linha 7, onde está `malloc`, vamos ter o seguinte resultado do `valgrind` quando compilarmos e executarmos ele novamente:

```

1 $ gcc -g -o null null.c
2 $ valgrind --tool=memcheck --leak-check=yes -q ./null
3 $ gcc -g -o null null.c
4 $ valgrind --tool=memcheck --leak-check=yes -q ./null
5 ==9029== Use of uninitialised value of size 4
6 ==9029==    at 0x8048459: main (null.c:8)
7 ==9029==

```

Reparem que agora ele já deduziu que o problema está na linha 8 (`null.c:8`), onde está sendo copiado um valor para uma variável não alocada.

Pequeno detalhe: nem toda String usa malloc/free

Apesar de mostrar e chorar as pitangas sobre `malloc` e `free` acima (ah vá, vocês gostaram das dicas em C), nem toda `String` em Ruby (pelo menos nas versões 1.9.x) são alocadas com `malloc`, diretamente no *heap*. Esses são os casos das chamadas “**Strings de heap**”. Existem também as “**Strings compartilhadas**”, que são `Strings` que apontam para outras, ou seja, quando utilizamos algo como `str2 = str1`, e vão apontar para o mesmo local.

Mas tem outro tipo de `Strings`. As com até 11 caracteres em máquinas 32 *bits* e 23 caracteres em máquinas 64 *bits*, são consideradas “**Strings embutidas**”, e tem, na estrutura interna de Ruby, um *array* de caracteres desses tamanhos respectivos já alocado, para onde a `String` é copiada direto, sem precisar da utilização de `malloc` e `free`, conseqüentemente, aumentando a velocidade. O nosso programa acima seria algo como:

```
1 #include <stdio.h>
2
3 int main() {
4     char str[15] = "hello world";
5     printf("%s\n",str);
6     return 0;
7 }
```

Fica até mais simples, mas a sequência de caracteres fica “engessada” nos 15 caracteres. As `Strings` que ultrapassam esses limites são automaticamente criadas ou promovidas para `Strings de heap`, ou seja, usam `malloc/free`. Se você ficou curioso com os limites, pode compilar (compilado aqui com o GCC em um GNU/Linux) e rodar esse programa:

```
1 #include <stdio.h>
2 #include <limits.h>
3
4 int main() {
5     printf("%d bits: %d bytes de comprimento\n",__WORDSIZE,((int)((sizeof(unsig\
6 nged int)*3)/sizeof(char)-1)));
7 }
```

O resultado vai ser algo como:

```
1 32 bits: 11 bytes de comprimento
```

Como curiosidade, essa é a estrutura que cuida de `Strings` no código de Ruby, `RString`:

```

1  struct RString {
2
3      struct RBasic basic;
4
5      union {
6          struct {
7              long len;
8              char *ptr;
9              union {
10                 long capa;
11                 VALUE shared;
12             } aux;
13         } heap;
14
15         char ary[RSTRING_EMBED_LEN_MAX + 1];
16     } as;
17 };

```

Se repararmos na primeira union definida, podemos ver que é ali que é gerenciado se vai ser utilizada uma String de *heap* ou embutida. Lembrem-se (ou saibam) que unions em C permitem que sejam armazenados vários tipos dentro dela, mas permite acesso a apenas um deles por vez. Esse programa aqui vai produzir um efeito indesejado, pois é atribuído um valor no primeiro membro e logo após no segundo membro, que *sobreescreve* o valor do primeiro, deixando ele totalmente maluco no caso da conversão para um int:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  union data {
6      int id;
7      char name[20];
8  };
9
10 int main() {
11     union data d;
12     d.id = 1;
13     strcpy(d.name, "taq");
14     printf("%d %s\n", d.id, d.name);
15     return 0;
16 }

```

Rodando o programa, temos algo como isso:

```
1 $ ./union
2 7430516 taq
```

Agora, se utilizarmos cada membro da union **de cada vez**, temos o comportamento esperado:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 union data {
6     int id;
7     char name[20];
8 };
9
10 int main() {
11     union data d;
12     d.id = 1;
13     printf("%d\n",d.id);
14     strcpy(d.name,"taq");
15     printf("%s\n",d.name);
16     return 0;
17 }
```

Rodando o programa:

```
1 $ ./union2
2 1
3 taq
```

Unit testing

Se você for usar Rails e não aprender a usar os recursos de testes do *framework*, que já vem todo estruturado, estará relegando um ganho de produtividade muito grande.

Testes unitários são meios de testar e depurar pequenas partes do seu código, para verificar se não tem alguma coisa errada acontecendo, “modularizando” a checagem de erros. Um sistema é feito de várias “camadas” ou “módulos”, e os testes unitários tem que ser rodados nessas camadas.

Vamos usar de exemplo uma calculadora que só tem soma e subtração, então vamos fazer uma classe para ela, no arquivo `calc.rb`:

```
1 class Calculadora
2   def soma(a,b)
3     a+b
4   end
5
6   def subtrai(a,b)
7     a-b
8   end
9 end
```

E agora o nosso teste propriamente dito, nos moldes das versões 1.8.x de Ruby:

```
1 require "test/unit"
2 require_relative "calc"
3
4 class TesteCalculadora < Test::Unit::TestCase
5   def setup
6     @calculadora = Calculadora.new
7   end
8
9   def test_adicao
10    assert_equal(2,@calculadora.soma(1,1),"1+1=2")
11  end
12
13  def test_subtracao
14    assert_equal(0,@calculadora.subtrai(1,1),"1-1=0")
15  end
16
17  def teardown
18    @calculadora = nil
19  end
20 end
```

Rodando os testes:

```
1 Run options:
2
3 # Running tests:
4 ..
5
6 Finished tests in 0.000438s, 4562.1062 tests/s, 4562.1062 assertions/s.
7
8 2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

Que é o resultado esperado quando todos os testes passam. Algumas explicações do arquivo de teste:

- A classe é estendida de `Test::Unit::TestCase`, o que vai “dedurar” que queremos executar os testes contidos ali.
- Temos o método `setup`, que é o “construtor” do teste, e vai ser chamado para todos os testes, não somente uma vez.
- Temos o método `teardown`, que é o “destrutor” do teste, e vai liberar os recursos alocados através do `setup`.
- Temos as asserções, que esperam que o seu tipo combine com o primeiro argumento, executando o teste especificado no segundo argumento, usando o terceiro argumento como uma mensagem de ajuda se por acaso o teste der errado.

Para demonstrar uma falha, faça o seu código de subtração ficar meio maluco, por exemplo, retornando o resultado mais 1, e rode os testes novamente:

```
1 $ ruby testcalc1.rb
2 Run options:
3
4 # Running tests:
5
6 .F
7
8 Finished tests in 0.000642s, 3116.3813 tests/s, 3116.3813 assertions/s.
9
10 1) Failure:
11 test_subtracao(TesteCalculadora) [testcalc1.rb:14]:
12 1-1=0.
13 <0> expected but was
14 <1>.
15
16 2 tests, 2 assertions, 1 failures, 0 errors, 0 skips
```

Além de `assert_equal`, temos várias outras asserções:

- `assert_nil`
- `assert_not_nil`
- `assert_not_equal`
- `assert_instance_of`
- `assert_kind_of`
- `assert_match`
- `assert_no_match`
- `assert_same`
- `assert_not_same`

Vamos incluir algumas outras no nosso arquivo:

```
1 require "test/unit"
2 require_relative "calc"
3
4 class TesteCalculadora < Test::Unit::TestCase
5   def setup
6     @calculadora = Calculadora.new
7   end
8
9   def test_objeto
10    assert_kind_of(Calculadora, @calculadora)
11    assert_match(/^d$/, @calculadora.soma(1,1).to_s)
12    assert_respond_to(@calculadora, :soma)
13    assert_same(@calculadora, @calculadora)
14  end
15
16  def test_objetos
17    assert_operator(@calculadora.soma(1,1), :>, @calculadora.soma(1,0))
18  end
19
20  def test_adicao
21    assert_equal(2, @calculadora.soma(1,1), "1+1=2")
22  end
23
24  def test_subtracao
25    assert_equal(0, @calculadora.subtrai(1,1), "1-1=0")
26  end
27
28  def teardown
29    @calculadora = nil
30  end
31 end
```

Rodando os novos testes:

```
1 $ ruby testcalc2.rb
2 Run options:
3
4 # Running tests:
5
6 ....
7
8 Finished tests in 0.000577s, 6934.5964 tests/s, 13869.1927 assertions/s.
9
10 4 tests, 8 assertions, 0 failures, 0 errors, 0 skips
```

Modernizando os testes

A partir da versão 1.9.x de Ruby, podemos contar com o *framework* de testes Minitest, e podemos reescrever nosso teste da calculadora dessa forma, definida no arquivo `minitest1.rb`:

```
1 require "minitest/autorun"
2 require_relative "calc"
3
4 class TesteCalculadora < Minitest::Unit::TestCase
5   def setup
6     @calculadora = Calculadora.new
7   end
8
9   def teardown
10    @calculadora = nil
11  end
12
13  def test_objeto
14    assert_kind_of(Calculadora, @calculadora)
15    assert_match(/^ \d$/, @calculadora.soma(1,1).to_s)
16    assert_respond_to(@calculadora, :soma)
17    assert_same(@calculadora, @calculadora)
18  end
19
20  def test_objetos
21    assert_operator(@calculadora.soma(1,1), :>, @calculadora.soma(1,0))
22  end
23
24  def test_adicao
25    assert_equal(2, @calculadora.soma(1,1), "1+1=2")
26  end
27
28  def test_subtracao
29    assert_equal(0, @calculadora.subtrai(1,1), "1-1=0")
```

```
30     end
31 end
```

Mas que? Só mudou de onde herdávamos de `Test::Unit::TestCase` e agora é `Minitest::Unit::TestCase`?

Randomizando os testes

Qual a vantagem? Antes de mais nada, vamos rodar o teste para ver o resultado:

```
1 $ ruby minitest1.rb
2 Run options: --seed 47074
3
4 # Running tests:
5
6 ....
7
8 Finished tests in 0.000556s, 7193.2484 tests/s, 14386.4968 assertions/s.
9
10 4 tests, 8 assertions, 0 failures, 0 errors, 0 skips
```

Reparem em `--seed 47074`. Ali é indicado que os testes são executados em ordem randômica, prevenindo a sua suíte de testes de ser executada dependente da ordem dos testes, o que ajuda a prevenir algo chamado de “*state leakage*” (“vazamento de estado”) entre os testes. Os testes tem que ser executados independente de sua ordem, e para isso o `Minitest` gera uma *seed* randômica para a execução dos testes. Se precisarmos executar os testes novamente com a mesma *seed*, já que ela vai ser alterada a cada vez que executamos os testes, podemos utilizar:

```
1 $ ruby minitest1.rb --seed 47074
```

Testando com specs

Também podemos testar utilizando *specs*, no estilo do `RSpec`, reescrevendo o código dessa maneira:

```
1  # encoding: utf-8
2  require "minitest/autorun"
3  require_relative "calc"
4
5  describe "Calculadora" do
6    before do
7      @calculadora = Calculadora.new
8    end
9
10   after do
11     @calculadora = nil
12   end
13
14   describe "objeto" do
15     it "deve ser do tipo de Calculadora" do
16       @calculadora.must_be_kind_of Calculadora
17     end
18     it "deve ter um método para somar" do
19       @calculadora.must_respond_to :soma
20     end
21     it "deve ter um método para subtrair" do
22       @calculadora.must_respond_to :subtrai
23     end
24   end
25
26   describe "soma" do
27     it "deve ser igual a 2" do
28       @calculadora.soma(1,1).must_equal 2
29     end
30   end
31
32   describe "subtração" do
33     it "deve ser igual a 0" do
34       @calculadora.subtrai(1,1).must_equal 0
35     end
36   end
37 end
```

Agora já mudou bastante! Podemos usar alguns atalhos como `let`, ao invés do método `before` (mostrando aqui só o primeiro teste):

```

1 # encoding: utf-8
2 require "minitest/autorun"
3 require_relative "calc"
4
5 describe "Calculadora" do
6   let(:calculadora) { Calculadora.new }
7
8   describe "objeto" do
9     it "deve ser do tipo de Calculadora" do
10       calculadora.must_be_kind_of Calculadora
11     end
12   ...

```

Podemos pular algum teste, utilizando skip:

```

1 it "deve ter um método para multiplicar" do
2   skip "ainda não aprendi como multiplicar"
3   calculadora.must_respond_to :multiplicar
4 end

```

Benchmarks

O Minitest já vem com recursos de *benchmarks*:

```

1 require "minitest/benchmark"
2 ...
3 describe "benchmarks" do
4   bench_performance_linear "primeiro algoritmo", 0.001 do |n|
5     100.times do |v|
6       calculadora.soma(n,v)
7     end
8   end
9
10  bench_performance_linear "segundo algoritmo", 0.001 do |n|
11    100.times do |v|
12      calculadora.soma(v,n)
13    end
14  end
15 end
16
17 $ ruby minitest3.rb
18 Calculadora::benchmarks          1          10          100          1000          10000
19 bench_primeiro_algoritmo 0.000084 0.000071 0.000065 0.000061 0.000060
20 bench_segundo_algoritmo 0.000070 0.000061 0.000059 0.000059 0.000059

```

Mocks

Temos um sistema básico e fácil para utilizar [mocks](#)³⁵, onde podemos simular o comportamento de um objeto complexo, ainda não acessível ou construído ou impossível de ser incorporado no teste. Um mock é recomendado se³⁶:

- Gera resultados não determinísticos (ou seja, que exibem diferentes comportamentos cada vez que são executados)
- Tem estados que são difíceis de criar ou reproduzir (por exemplo, erro de comunicação da rede)
- É lento (por exemplo, um banco de dados completo que precisa ser inicializado antes do teste)
- Ainda não existe ou pode ter comportamento alterado
- Teriam que adicionar informações e métodos exclusivamente para os testes (e não para sua função real)

Existem algumas *gems* para utilizarmos *mocks*, como a Mocha (<https://github.com/freerange/mocha>³⁷), que tem vários recursos interessantes, mas com o Minitest grande parte do que precisamos já está pronto.

Primeiro, vamos alterar `calc.rb` para incluir um método chamado `media`, que vai receber e calcular a média de uma coleção:

```
1 class Calculadora
2   def soma(a,b)
3     a+b
4   end
5
6   def subtrai(a,b)
7     a-b
8   end
9
10  def media(colecao)
11    val = colecao.valores
12    val.inject(:+)/val.size.to_f
13  end
14 end
```

E agora vamos utilizar um `Mock` para simular um objeto de coleção (apesar que poderia facilmente ser um `Array`). Para isso, vamos ver agora o teste, mostrando somente o método que utiliza o `Mock`:

³⁵http://pt.wikipedia.org/wiki/Mock_Object

³⁶http://pt.wikipedia.org/wiki/Mock_Object

³⁷<https://github.com/freerange/mocha>

```
1 ...
2 describe "média" do
3   it "deve ser igual a 2" do
4     colecao = MiniTest::Mock.new
5     colecao.expect :valores, [1,2,3]
6     calculadora.media(colecao)
7     colecao.verify
8   end
9 end
```

“Falsificamos” um objeto, com um método chamado `valores`, que retorna um `Array` de 3 `Fixnum`'s: `[1,2,3]`.

Stubs

Também podemos ter `stubs`³⁸, que podem ser utilizados como substitutos temporários de métodos que demorem muito para executar, consumam muito processamento, etc. No caso dos `Stubs` do `Minitest`, eles duram dentro e enquanto durar o bloco que foram definidos:

```
1 describe "soma maluca" do
2   it "deve ser igual a 3" do
3     calculadora.stub :soma, 3 do
4       calculadora.soma(1,1).must_equal 3
5     end
6   end
7 end
```

Esse exemplo foi para efeitos puramente didáticos - e inúteis, do ponto de vista de uma calculadora que iria retornar um valor totalmente inválido - mas serve para mostrar como podemos fazer uso de `stubs`.

Expectations

Algumas das `expectations`³⁹ do `Minitest`. Para testarmos uma condição inversa, na maioria das vezes é só trocar `must` para `wont`, por exemplo, `must_be` por `wont_be`:

- `must_be` - Testa uma condição comparando o valor retornado de um método:

```
1 10.must_be :<, 20
```

- `must_be_empty` - Deve ser vazio:

³⁸<http://pt.wikipedia.org/wiki/Stub>

³⁹<http://www.ruby-doc.org/stdlib-1.9.3/libdoc/minitest/spec/rdoc/MiniTest/Expectations.html>

```
1 [] .must_be_empty
```

- **must_be_instance_of** - Deve ser uma instância de uma classe:

```
1 "oi".must_be_instance_of String
```

- **must_be_kind_of** - Deve ser de um determinado tipo:

```
1 1.must_be_kind_of Numeric
```

- **must_be_nil** - Deve ser nulo:

```
1 a = nil
2 a.must_be_nil
```

- **must_be_same_as** - Deve ser o mesmo objeto:

```
1 a = "oi"
2 b = a
3 a.must_be_same_as b
```

- **must_be_silent** - O bloco não pode mandar nada para stdout ou stderr:

```
1 -> {}.must_be_silent
2 => true
3 -> { puts "oi" }.must_be_silent
4 1) Failure:
5 test_0002_should be silent(Test) [minitest.rb:10]:
6 In stdout.
```

- **must_be_within_delta(exp,act,delta,msg)** - Compara Floats, verificando se o valor de exp tem uma diferença de no máximo delta de act, comparando se delta é maior que o valor absoluto de exp-act ($\text{delta} > (\text{exp} - \text{act}).\text{abs}$):

```
1 1.01.must_be_within_delta 1.02, 0.1
2 => true
3 1.01.must_be_within_delta 1.02, 0.1
4 Expected |1.02 - 1.01| (0.010000000000000009) to be < 0.009
```

- **must_be_within_epsilon(exp,act,epsilon,msg)** - Similar ao delta, mas epsilon é uma medida de erro relativa aos pontos flutuantes. Compara utilizando **must_be_within_delta**, calculando delta como o valor mínimo entre exp e act, vezes epsilon (`must_be_within_delta exp, act, [exp,act].min*epsilon`).

- **must_equal** - Valores devem ser iguais. Para Floats, use `must_be_within_delta` explicada logo acima.

```
1 a.must_equal b
```

- **must_include** - A coleção deve incluir o objeto:

```
1 (0..10).must_include 5
```

- **must_match** - Deve “casar”:

```
1 "1".must_match /\d/
```

- **must_output(stdout,stderr)** - Deve imprimir determinado o resultado esperado em stdout ou stderr. Para testar somente em stderr, envie nil no primeiro argumento:

```
1 -> { puts "oi" }.must_output "oi\n"
2 => true
3 -> { }.must_output "oi\n"
4 1) Failure:
5 test_0004_should output(Test) [minitest.rb:20]:
6 In stdout.
```

- **must_raise** - Deve disparar uma Exception:

```
1 -> { 1+"um" }.must_raise TypeError
2 => true
3 -> { 1+1 }.must_raise TypeError
4 1) Failure:
5 test_0005_should raises an exception(Test) [minitest.rb:25]:
6 TypeError expected but nothing was raised.
```

- **must_respond_to** - Deve responder à um determinado método:

```
1 "oi".must_respond_to :upcase
```

- **must_send** - Deve poder ser enviado determinado método com argumentos:

```
1 must_send ["eustáquio", :slice, 3, 3]
```

- **must_throw** - Deve disparar um throw:

```
1 ->{ throw :custom_error }.must_throw :custom_error
```

Já deixando claro que existe uma pequena grande diferença entre `kind_of?` (tipo de) e `instance_of?` (instância de). Deêm uma olhada nesse código:

```
1 class A; end
2 class B < A; end
3 b = B.new
4 b.instance_of?(B)
5 => true
6 b.instance_of?(A)
7 => false
8 b.kind_of?(B)
9 => true
10 b.kind_of?(A)
11 => true
12 A===b
13 => true
14 B===b
15 => true
```

Dá para perceber que `===`, para classes, é um *alias* de `kind_of?`.

Testes automáticos

Nada mais chato do que ficar rodando os testes manualmente após alterarmos algum conteúdo. Para evitar isso, temos algumas ferramentas como o [Guard](#)⁴⁰, que automatizam esse processo. Podemos instalar as seguintes *gems* para utilizar Guard e Minitest:

```
1 gem install guard
2 gem install guard-minitest
```

Após isso, podemos executar:

```
1 guard init minitest
```

Deixar o arquivo `Guardfile` criado dessa maneira:

```
1 # minitest guard file
2
3 guard 'minitest' do
4   watch(%r|^spec/(.*)_spec\.rb|)
5   watch(%r|^lib/(.*)(([/]+)\.rb|) {|m| "spec/#{m[1]}#{m[2]}_spec.rb"})
6   watch(%r|^spec/spec_helper\.rb|) {"spec"}
7 end
```

Gravar nossos testes em um diretório chamado `spec` (viram ele referenciado ali em cima?), em arquivos chamados `*_spec.rb` (também viram a máscara `*_spec.rb` ali?) e finalmente rodar o comando `guard`:

⁴⁰<https://github.com/guard/guard>

- 1 `$ guard`
- 2 `14:10:44 - INFO - Guard uses Libnotify to send notifications.`
- 3 `14:10:44 - INFO - Guard uses TerminalTitle to send notifications.`

Os testes encontrados vão ser avaliados sempre que algum arquivo for alterado.

Criando Gems

Podemos criar *gems* facilmente, desde escrevendo os arquivos de configuração “na unha”, até utilizando a *gem bundle*, que provavelmente já se encontra instalada no sistema.

Criando a gem

Vamos construir uma *gem* para “aportuguesar” os métodos `even?` e `odd?`, traduzindo-os respectivamente para `par?` e `impar?`. Para criar a nova *gem*, chamada `portnum`, podemos digitar:

```
1 $ bundle gem portnum
2 Bundling your gems. This may take a few minutes on first run.
3 create portnum/Gemfile
4 create portnum/Rakefile
5 create portnum/.gitignore
6 create portnum/portnum.gemspec
7 create portnum/lib/portnum.rb
8 create portnum/lib/portnum/version.rb
9 Initializing git repo in /home/aluno/gem/portnum
```

Esse comando gera a seguinte estrutura de diretório/arquivos, inclusive já dentro de um repositório do Git:

```
1 $ cd portnum/
2 $ ls -lah
3 total 32K
4 drwxr-xr-x 4 taq taq 4,0K 2011-07-14 17:40 .
5 drwxr-xr-x 3 taq taq 4,0K 2011-07-14 17:40 ..
6 -rw-r--r-- 1 taq taq 91    2011-07-14 17:40 Gemfile
7 drwxr-xr-x 7 taq taq 4,0K 2011-07-14 17:40 .git
8 -rw-r--r-- 1 taq taq 33    2011-07-14 17:40 .gitignore
9 drwxr-xr-x 3 taq taq 4,0K 2011-07-14 17:40 lib
10 -rw-r--r-- 1 taq taq 681   2011-07-14 17:40 portnum.gemspec
11 -rw-r--r-- 1 taq taq 28    2011-07-14 17:40 Rakefile
```

O ponto-chave é o arquivo `portnum.gemspec`:

```
1 # -*- encoding: utf-8 -*-
2 $.push File.expand_path("../lib", __FILE__)
3 require "portnum/version"
4
5 Gem::Specification.new do |s|
6   s.name = "portnum"
7   s.version = Portnum::VERSION
8   s.authors = ["TODO: Write your name"]
9   s.email = ["TODO: Write your email address"]
10  s.homepage = ""
11  s.summary = %q{TODO: Write a gem summary}
12  s.description = %q{TODO: Write a gem description}
13  s.files = `git ls-files`.split("\n")
14  s.test_files = `git ls-files -- {test,spec,features}/*`.split("\n")
15  s.executables = `git ls-files -- bin/*`.split("\n").map{ |f| File.basename\
16 e(f) }
17   s.require_paths = ["lib"]
18 end
```

Temos que preencher com os dados necessários:

```
1 # -*- encoding: utf-8 -*-
2 $.push File.expand_path("../lib", __FILE__)
3 require "portnum/version"
4
5 Gem::Specification.new do |s|
6   s.name = "portnum"
7   s.version = Portnum::VERSION
8   s.authors = ["Eustaquio Rangel"]
9   s.email = ["taq@bluefish.com.br"]
10  s.homepage = "http://eustaquiorangel.com/portnum"
11  s.summary = %q{Aportuguesamento de números}
12  s.description = %q{Adiciona os métodos par? e impar? na classe Numeric}
13  s.files = `git ls-files`.split("\n")
14  s.test_files = `git ls-files -- {test,spec,features}/*`.split("\n")
15  s.executables = `git ls-files -- bin/*`.split("\n").map{ |f| File.basename\
16 e(f) }
17   s.require_paths = ["lib"]
18 end
```

Dentro do diretório lib, se encontram os seguintes arquivos:

```
1 $ ls -lah lib
2 total 16K
3 drwxr-xr-x 3 taq taq 2011-07-14 17:40 .
4 drwxr-xr-x 4 taq taq 2011-07-14 17:40 ..
5 drwxr-xr-x 2 taq taq 2011-07-14 17:40 portnum
6 -rw-r--r-- 1 taq taq 2011-07-14 17:40 portnum.rb
7
8 $ ls -lah lib/portnum
9 total 12K
10 drwxr-xr-x 2 taq taq 4,0K 2011-07-14 17:40 .
11 drwxr-xr-x 3 taq taq 4,0K 2011-07-14 17:40 ..
12 -rw-r--r-- 1 taq taq 39 2011-07-14 17:40 version.rb
```

Dentro do arquivo `version.rb`, temos:

```
1 $ cat lib/portnum/version.rb
2 module Portnum
3   VERSION = "0.0.1"
4 end
```

Que vai definir o número de versão da nossa *gem*. Dentro do arquivo `portnum.rb`, temos:

```
1 $ cat lib/portnum.rb
2 require "portnum/version"
3
4 module Portnum
5   # Your code goes here...
6 end
```

Esse é o código que vai ser carregado quando a *gem* for requisitada. Vamos alterar a classe `Numeric` lá, para implementar os nossos dois métodos:

```
1 require "#{File.expand_path(File.dirname(__FILE__))}/portnum/version"
2
3 class Numeric
4   def par?
5     self%2==0
6   end
7   def impar?
8     self%2==1
9   end
10 end
```

Testando a gem

Antes de construir nossa *gem*, vamos criar alguns testes no diretório `test`, que deverá ser criado:

```
1 require "test/unit"
2 require "#{File.expand_path(File.dirname(__FILE__))}/../lib/portnum.rb"
3
4 class PortNumTest < Test::Unit::TestCase
5   def test_par
6     assert_respond_to 1, :par?
7   end
8
9   def test_par_ok
10    assert 2.par?
11    assert !1.par?
12  end
13
14  def test_impar
15    assert_respond_to 1, :impar?
16  end
17
18  def test_impar_ok
19    assert 1.impar?
20    assert !2.impar?
21  end
22 end
```

Rodando os testes:

```
1 $ ruby test/portnumtest.rb
2 Loaded suite test/portnumtest
3 Started
4 .....
5 Finished in 0.000473 seconds.
6 5 tests, 7 assertions, 0 failures, 0 errors, 0 skips
7 Test run options: --seed 31305
```

Podemos criar uma *task* em nosso Rakefile para executar nossos testes:

```
1 require 'bundler/gem_tasks'
2 require 'rake'
3 require 'rake/testtask'
4
5 Rake::TestTask.new(:test) do |test|
6   test.libs << 'lib' << 'test'
7   test.pattern = 'test/*.rb'
8 end
9
10 $ rake test
```

```
11 Loaded suite
12 Started
13 .....
14 Finished in 0.000596 seconds.
15 5 tests, 7 assertions, 0 failures, 0 errors, 0 skips
16 Test run options: --seed 663
```

Construindo a gem

Agora que verificamos que tudo está ok, vamos construir a nossa *gem*:

```
1 $ rake build
2 portnum 0.0.1 built to pkg/portnum-0.0.1.gem
3 $ ls -lah pkg/
4 total 12K
5 drwxr-xr-x 2 taq taq 4,0K 2011-07-14 19:42 .
6 drwxr-xr-x 6 taq taq 4,0K 2011-07-14 19:42 ..
7 -rw-r--r-- 1 taq taq 4,0K 2011-07-14 19:42 portnum-0.0.1.gem
```

Olha lá a nossa *gem*! Agora vamos instalá-la:

```
1 $ rake install
2 portnum 0.0.1 built to pkg/portnum-0.0.1.gem
3 portnum (0.0.1) installed
```

Testando se deu certo:

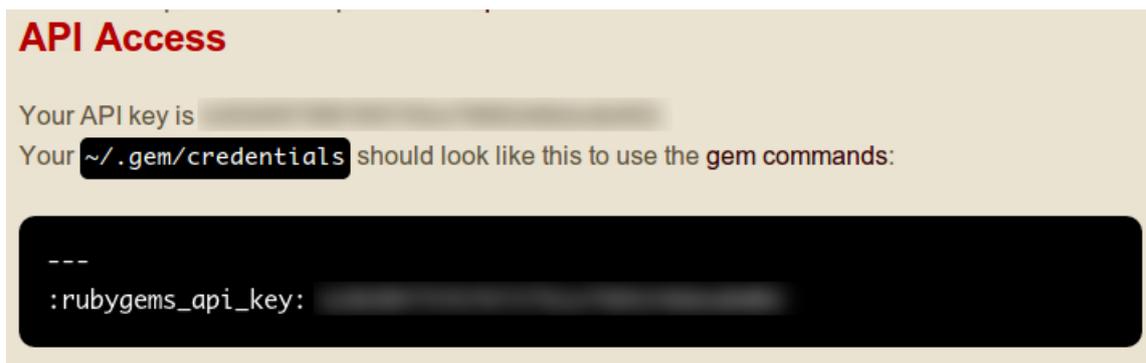
```
1 $ irb
2 require "portnum"
3 => true
4 1.par?
5 => false
6 1.impar?
7 => true
```

Publicando a gem

Podemos publicar a gem facilmente para o [RubyGems.org](http://rubygems.org)⁴¹, que é o repositório oficial de *gems* para Ruby. Primeiro temos que criar uma conta lá, e indo em <https://rubygems.org/profile/edit>⁴² e salvar a nossa chave da API para um arquivo YAML em `~/ .gem/credentials`:

⁴¹<http://rubygems.org>

⁴²<https://rubygems.org/profile/edit>



Credenciais da gem

Aí é só usar o comando `gem push`:

```
1 $ gem push portnum-0.0.1.gem
```

Se quisermos fazer os seguintes passos:

1. Executar o build
2. Criar uma tag no git e fazer um push para o repositório de código
3. Publicar a *gem* no `RubyGems.org`

podemos utilizar:

```
1 $ rake release
2 portnum 0.0.1 built to pkg/portnum-0.0.1.gem
3 Tagged v0.0.1
4 ...
```

Para ver todas as tasks que o Rake suporta:

```
1 $ rake -T
2 rake build # Build portnum-0.0.1.gem into the pkg directory
3 rake install # Build and install portnum-0.0.1.gem into system gems
4 rake release # Create tag v0.0.1 and build and push portnum-0.0.1.gem to R...
5 rake test # Run tests
```

Extraindo uma gem

Podemos extrair o código (com toda a estrutura de diretórios) contido em uma *gem* utilizando o comando `gem` com a opção `unpack`:

```
1 $ gem unpack portnum
```

Ou, no caso de não ter as *gems* instaladas, utilizando a ferramenta GNU `tar`:

```
1 $ tar xvf portnum-0.0.1.gem data.tar.gz
2 $ tar tvf data.tar.gz
```

Assinando uma gem

Em razão de um problema de comprometimento do [RubyGems](http://rubygems.org)⁴³ em Janeiro de 2013, os autores de *gems* foram instruídos a assinarem as suas *gems* usando um certificado auto-assinado baseado em [RSA](http://pt.wikipedia.org/wiki/RSA)⁴⁴, de forma que quando instaladas ou atualizadas, as *gems* possam ter a sua integridade verificada.

Criando um certificado

Para criar o seu certificado, digite o seguinte comando, trocando `<seu_email>` para o email que deseja que esteja associado ao certificado, digitando a senha do certificado (não se esqueça dela!) duas vezes:

```
1 $ gem cert --build <seu_email>
2 Passphrase for your Private Key:
3
4 Please repeat the passphrase for your Private Key:
5
6 Certificate: /home/taq/gem-public_cert.pem
7 Private Key: /home/taq/gem-private_key.pem
8 Don't forget to move the key file to somewhere private!
```

Podemos ver que foram criados dois arquivos no diretório corrente:

- `gem-public_cert.pem`
- `gem-private_cert.pem`

É uma boa idéia movermos esses arquivos para um diretório específico, como por exemplo, `~/gemcert`:

⁴³<http://rubygems.org>

⁴⁴<http://pt.wikipedia.org/wiki/RSA>

```
1 $ mkdir ~/.gemcert
2
3 $ mv -v gem-p* ~/.gemcert/
4 "gem-private_key.pem" -> "/home/taq/.gemcert/gem-private_key.pem"
5 "gem-public_cert.pem" -> "/home/taq/.gemcert/gem-public_cert.pem"
```

Uma **grande** diferença entre esses arquivos é que o **private** tem que ficar bem guardado em segredo, sem divulgar ou entregar para alguém, para evitar que alguém se faça passar por você, enquanto o **public** pode e deve ser publicado para que as pessoas possam conferir a assinatura da gem que usa esse certificado, no velho esquema de chaves públicas e privadas.

Adaptando a gem para usar o certificado

Vamos pegar como exemplo uma gem que mantenho, a [Traquitana](#)⁴⁵. Para indicar que ela vai utilizar o meu certificado, vou inserir as seguintes linhas no final do arquivo `traquitana.gemspec`:

```
1 gem.signing_key = '/home/taq/.gemcert/gem-private_key.pem'
2 gem.cert_chain = ['gem-public_cert.pem']
```

Isso vai indicar que o arquivo **private** vai ser utilizado para assinar a gem, e o arquivo **public** vai ser utilizado para conferir a assinatura. Podemos publicar nosso arquivo **public** em algum lugar na web, mas vamos facilitar e distribuí-lo junto com o código da nossa gem. Para isso, vá para o diretório onde está o arquivo `.gemspec` da gem (no caso acima, o `traquitana.gemspec`) e copie o arquivo **public** do seu diretório `~/.gemcert` (ou de onde você armazenou os arquivos):

```
1 $ cp ~/.gemspec/gem-public_cert.pem .
```

Construindo e publicando a gem assinada

Agora podemos construir a gem assinada, utilizando o `rake build` como vimos acima, com a diferença que agora vai ser solicitada a senha utilizada na criação do certificado:

```
1 $ rake build
2 Enter PEM pass phrase:
3 traquitana 0.0.23 built to pkg/traquitana-0.0.23.gem.
```

Podemos também utilizar `rake release` para fazer o processo completo, como demonstrado um pouco antes, sem problemas.

Utilizando a gem assinada

Com a gem assinada e publicada, agora podemos instalá-la ou atualizá-la pedindo para que seja verificada no momento da operação selecionada. Para isso, vamos precisar importar os certificados **públicos** disponibilizados pelos desenvolvedores das gems e utilizar a opção `-P HighSecurity`. Se, por exemplo, eu tentar atualizar a gem em um computador que não tem o certificado importado, não conseguindo verificar a integridade da gem, vai acontecer isso:

⁴⁵<https://github.com/taq/traquitana>

```
1 $ gem update traquitana -P HighSecurity
2 Updating installed gems
3 Updating traquitana
4 Fetching: traquitana-0.0.23.gem (100%)
5 ERROR: While executing gem ... (Gem::Security::Exception)
6   root cert /CN=taq/DC=eustaquiorangel/DC=com is not trusted
```

Vamos dar uma olhada nos certificados que temos disponíveis:

```
1 $ gem cert --list
```

Não foi retornado nada aqui, então vamos importar o certificado disponibilizado com a gem, que nesse caso, se encontra disponível em https://raw.githubusercontent.com/taq/traquitana/master/gem-public_cert.pem⁴⁶, de onde vamos fazer *download* para um arquivo local, importar o certificado e logo depois apagar o arquivo local:

```
1 $ curl https://raw.githubusercontent.com/taq/traquitana/master/gem-public_cer\
2 t.pem > cert
3
4 $ gem cert --add cert
5 Added '/CN=taq/DC=eustaquiorangel/DC=com'
6
7 $ rm cert
8
9 $ gem cert --list
10 /CN=taq/DC=eustaquiorangel/DC=com
```

Com o certificado instalado, vamos tentar atualizar a gem novamente com a opção de verificação:

```
1 $ gem update traquitana -P HighSecurity
2 Updating installed gems
3 Updating traquitana
4 Successfully installed traquitana-0.0.23
5 Parsing documentation for traquitana-0.0.23
6 Installing ri documentation for traquitana-0.0.23
7 Installing darkfish documentation for traquitana-0.0.23
8 Done installing documentation for traquitana after 0 seconds
9 Gems updated: traquitana
```

Agora funcionou tudo de acordo.

⁴⁶https://raw.githubusercontent.com/taq/traquitana/master/gem-public_cert.pem

**Dica**

Para utilizar a verificação com o [Bundler](http://bundler.io/)⁴⁷, podemos utilizar a opção `--trust-policy HighSecurity`, que funciona da mesma forma demonstrada acima. Por exemplo:

```
1 $ bundle install --trust-policy HighSecurity
```

⁴⁷<http://bundler.io/>

Gerando documentação

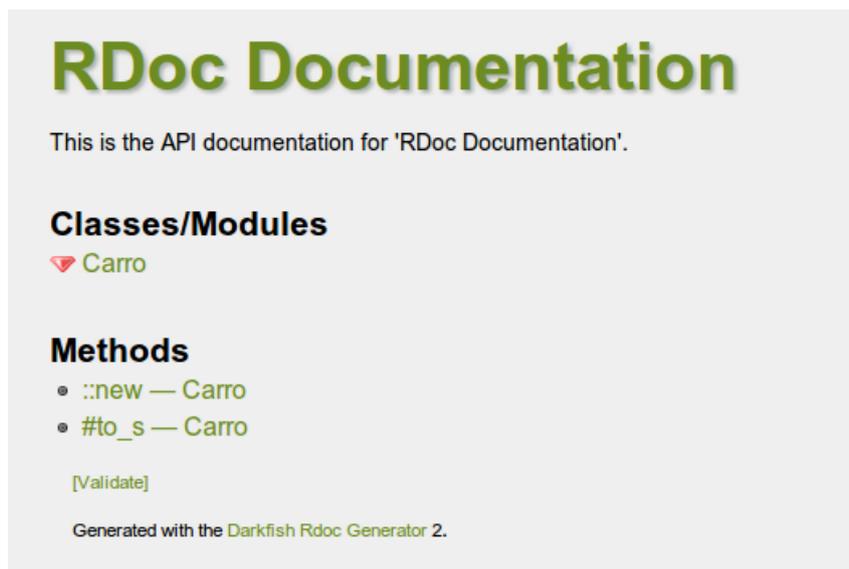
Vamos ver como podemos documentar o nosso código utilizando o `rdoc`, que é uma aplicação que gera documentação para um ou vários arquivos com código fonte em Ruby, interpretando o código e extraindo as definições de classes, módulos e métodos. Vamos fazer um arquivo com um pouco de código, usando nossos exemplos de carros:

```
1  # Essa é a classe base para todos os carros que vamos
2  # criar no nosso programa. A partir dela criamos carros
3  # de marcas específicas.
4  #
5  # Autor:: Eustáquio 'TaQ' Rangel
6  # Licença:: GPL
7  class Carro
8      attr_reader :marca, :modelo, :tanque
9      attr_accessor :cor
10
11     # Parâmetros obrigatórios para criar o carro
12     # Não se esqueça de que todo carro vai ter os custos de:
13     # * IPVA
14     # * Seguro obrigatório
15     # * Seguro
16     # * Manutenção
17     def initialize(marca, modelo, cor, tanque)
18         @marca = marca
19         @modelo = modelo
20         @cor = cor
21         @tanque = tanque
22     end
23
24     # Converte o carro em uma representação mais legível
25     def to_s
26         "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
27     end
28 end
```

Agora vamos rodar o `rdoc` nesse arquivo:

```
1 $ rdoc carro.rb
2 Parsing sources...
3 100% [ 1/ 1] carro.rb
4 Generating Darkfish format into doc...
5 Files: 1
6 Classes: 1
7 Modules: 0
8 Constants: 0
9 Attributes: 4
10 Methods: 2
11 (0 undocumented)
12 (0 undocumented)
13 (0 undocumented)
14 (4 undocumented)
15 (0 undocumented)
16
17 Total: 7 (4 undocumented)
18 42.86% documented
19 Elapsed: 0.1s
```

Isso vai produzir um diretório chamado `doc` abaixo do diretório atual, que vai conter um arquivo `index.html` com um conteúdo como esse:



Conteúdo do `index.html`

Clicando no link da classe `Carro`, vamos ter algo como:

Carro

Essa é a classe base para todos os carros que vamos criar no nosso programa. A partir dela criamos carros de marcas específicas.

Autor Eustáquio TaQ' Rangel

Licença GPL

Attributes

- marca^[R]
- modelo^[R]
- tanque^[R]
- cor^[RW]

Public Class Methods

new(marca, modelo, cor, tanque)
 Parâmetros obrigatórios para criar o carro Não se esqueça de que todo carro vai ter os custos de:

- IPVA
- Seguro obrigatório
- Seguro
- Manutenção

Public Instance Methods

to_s()
 Converte o carro em uma representação mais legível

Classe Carro

Pudemos ver algumas convenções para escrever a documentação. Os comentários são utilizados como as descrições das classes, módulos ou métodos. Podemos reparar que, se clicarmos no nome de algum método, o código-fonte desse método é mostrado logo abaixo, como em:

to_s()
 Converte o carro em uma representação mais legível

```
# File carro.rb, line 25
def to_s
  "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
end
```

Código fonte do método



Dica

Um detalhe muito importante é que se precisarmos gerar a documentação novamente sem alterar os fontes, devemos apagar o diretório onde ela foi gerada antes de rodar o rdoc novamente.

Algumas outras dicas de formatação:

- Texto do tipo *labeled lists*, que são listas com o suas descrições alinhadas, como no caso do autor e da licença do exemplo, são criados utilizando o valor e logo em seguida 2 dois pontos (: :), seguido da descrição.
- Listas de *bullets* são criadas usando asterisco (*) ou hífen (-) no começo da linha.

- Para listas ordenadas, temos que usar o número do item da lista seguido por um ponto (.).
- Cabeçalhos são gerados usando = para determinar o nível do cabeçalho, como:

```
1   = Primeiro nível
2   == Segundo nível
```

- Linhas podem ser inseridas usando três ou mais hifens.
- Negrito pode ser criado usando asteriscos (*) em volta do texto, como em **negrito**,
- Itálico pode ser criado com sublinhados (.) em volta do texto
- Fonte de tamanho fixo entre sinais de mais (+)
- Hyperlinks começando com `http:`, `mailto:`, `ftp:` e `www` são automaticamente convertidos. Também podemos usar o formato `texto[url]`.
- Nomes de classes, arquivos de código fonte, e métodos tem links criados do texto dos comentários para a sua descrição.

O processamento dos comentários podem ser interrompido utilizando - e retornado utilizando ++. Isso é muito útil para comentários que não devem aparecer na documentação.

Vamos ver nosso exemplo incrementado com todas essas opções e mais um arquivo novo, uma classe filha de Carro chamada Fusca, separando os dois arquivos em um diretório para não misturar com o restante do nosso código:

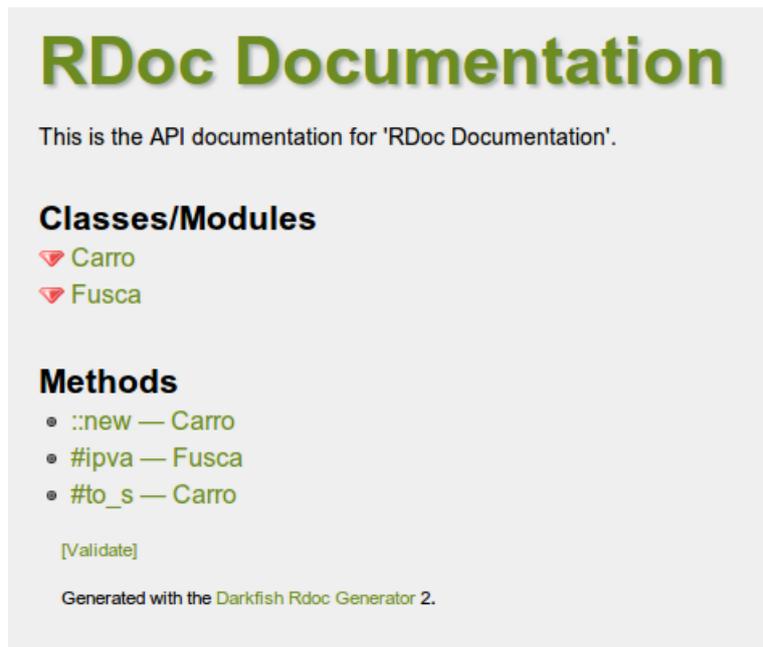
```
1  # = Classe
2  # Essa é a classe base para *todos* os carros que vamos
3  # criar no nosso programa. A partir dela criamos carros
4  # de _marcas_ específicas. Verique o método to_s dessa
5  # classe Carro para uma descrição mais legível.
6  # ---
7  #
8  # == Sobre o autor e licença
9  #
10 # Autor:: Eustáquio 'TaQ' Rangel
11 # Website:: http://eustaquiorangel.com
12 # Email:: mailto:naoteconto@eustaquiorangel.com
13 # Licença:: +GPL+ Clique aqui para ver mais[http://www.fsf.org]
14 #--
15 # Ei, ninguém deve ler isso.
16 #++
17 # Obrigado pela preferência.
18 class Carro
19     attr_reader :marca, :modelo, :tanque
20     attr_accessor :cor
21
22     # Parâmetros obrigatórios para criar o carro
23     # Não se esqueça de que todo carro vai ter os custos de:
24     # * IPVA
```

```
25     # * Seguro obrigatório
26     # * Seguro
27     # * Manutenção
28     def initialize(marca,modelo,cor,tanque)
29         @marca = marca
30         @modelo = modelo
31         @cor = cor
32         @tanque = tanque
33     end
34
35     # Converte o carro em uma representação mais legível
36     def to_s
37         "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
38     end
39 end
40
41 # Classe de um _vokinho_, derivada da classe Carro.
42 class Fusca < Carro
43     def ipva
44         false
45     end
46 end
```

Rodando o rdoc (prestem atenção que agora não especifico o arquivo):

```
1 $ rdoc
2 Parsing sources...
3 100% [ 2/ 2] fusca.rb
4 Generating Darkfish format into /home/taq/git/curso-ruby-rails/code/rdoc/doc.\
5 ..
6 Files: 2
7 Classes: 2
8 Modules: 0
9 Constants: 0
10 Attributes: 4
11 Methods: 3
12 (0 undocumented)
13 (0 undocumented)
14 (0 undocumented)
15 (4 undocumented)
16 (1 undocumented)
17 Total:
18 9 (5 undocumented)
19 44.44% documented
20 Elapsed: 0.1s
```

Vamos ter um resultado como esse:



RDoc Documentation

This is the API documentation for 'RDoc Documentation'.

Classes/Modules

- ♥ Carro
- ♥ Fusca

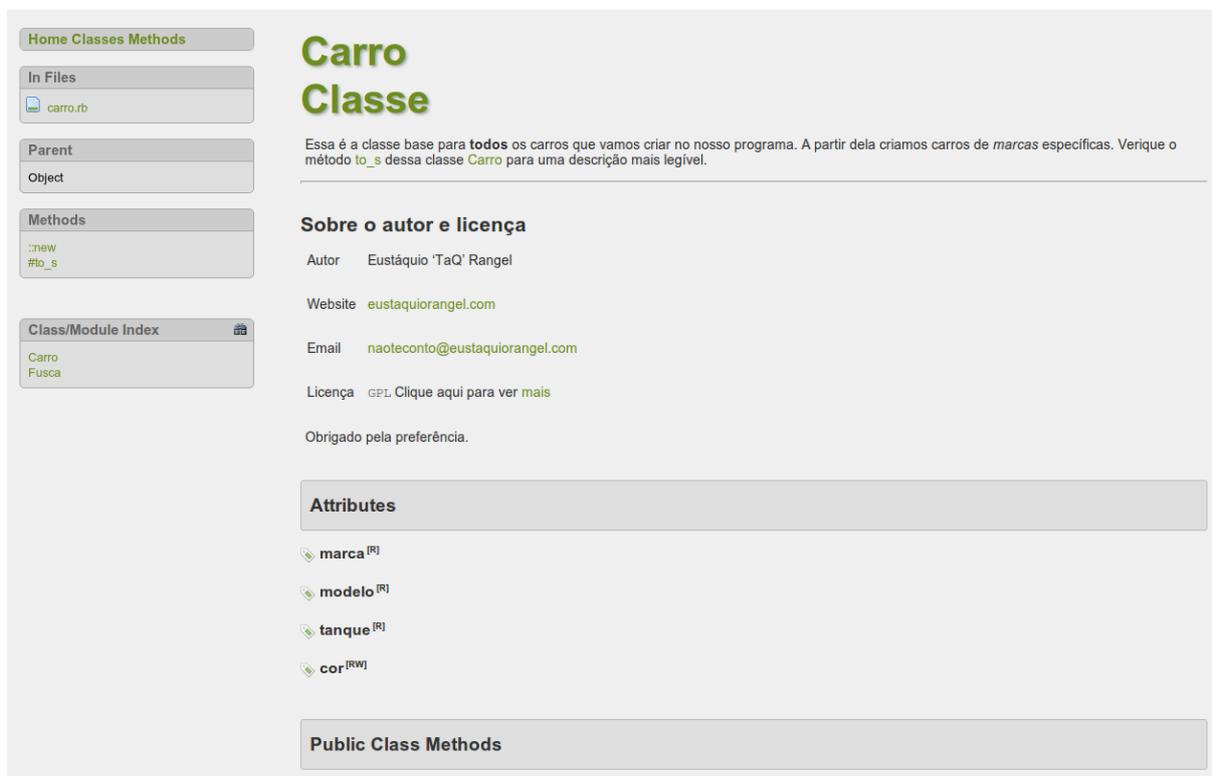
Methods

- `::new` — Carro
- `#ipva` — Fusca
- `#to_s` — Carro

[Validate]

Generated with the [Darkfish Rdoc Generator 2](#).

Classes e métodos



Carro
Classe

Essa é a classe base para **todos** os carros que vamos criar no nosso programa. A partir dela criamos carros de *marcas* específicas. Verique o método `to_s` dessa classe `Carro` para uma descrição mais legível.

Sobre o autor e licença

Autor `Eustáquio 'TaQ' Rangel`

Website eustaquiorangel.com

Email naoleconto@eustaquiorangel.com

Licença [GPL](#) [Clique aqui para ver mais](#)

Obrigado pela preferência.

Attributes

- ♥ `marca`^[R]
- ♥ `modelo`^[R]
- ♥ `tanque`^[R]
- ♥ `cor`^[RW]

Public Class Methods

Classe Carro



Novidades em Ruby 2.0

Agora o Rdoc entende [Markdown](#)⁴⁸. Para utilizar, devemos executar:

```
1 rdoc --markup markdown
```

E podemos deixar no diretório do projeto em um arquivo chamado `.doc_options`, para não ter que repetir toda vez, utilizando

```
1 rdoc --markup markdown --write-options
```

⁴⁸<http://daringfireball.net/projects/markdown/syntax>

Desafios

Desafio 1

A atribuição em paralelo mostra que primeiro o lado direito da expressão de atribuição é avaliado (ou seja, tudo à direita do sinal de igual) e somente após isso, os resultados são enviados para a esquerda, “encaixando” nos devidos locais, dessa maneira:

```
1 x, y = 1, 2
2 y, x = x, y
3 x
4 => 2
5 y
6 => 1
```

Desafio 2

Cada elemento da Hash é convertido em um Array para ser comparado. Por isso que podemos utilizar algo como `elemento1[1]`, onde no caso do primeiro elemento, vai ser convertido em `[:joao,33]`.

Desafio 3

Se você criou algo como:

```
1 v1 = "oi mundo"
2 v2 = Carro.new
3 v3 = 1
```

Isso significa que `v3` não vai apresentar a mensagem pois um `Fixnum` não aloca espaço na memória, que conseqüentemente não é processado pelo *garbage collector*.

Desafio 4

O código que utilizou `threads` manteve a sincronia da variável `res`, indicando no final a ordem em que foram terminando. O código que utilizou `processes`, não.

Desafio 5

Podemos atingir o mesmo comportamento usando `Hash` dessa forma:

```
1 # encoding: utf-8
2
3 str = <<FIM
4 texto para mostrar como podemos separar palavras do texto
5 para estatística de quantas vezes as palavras se repetem no
6 texto
7 FIM
8
9 p str.scan(/\w\p{Latin}+/).inject(Hash.new(0)) {|memo, word| memo[word] += 1; \
10 memo}
```

Desafio 6

Seguindo a [URL da documentação do método pack](#)⁴⁹ e analisando LA10A*, encontramos:

- L | Integer | 32-bit unsigned, native endian (uint32_t)
- A | String | arbitrary binary string (space padded, count is width)
- If the count is an asterisk (“*”), all remaining array elements will be converted.

Ou seja, estamos enviando um **inteiro (Fixnum)** (L), seguido de uma String com tamanho 10 (A10), seguido de uma String sem tamanho definido (A*), assumindo o resto dos *bytes*, que é o resultado do uso de Marshal na Hash. Mais informações na [URL da documentação de unpack](#)⁵⁰

Desafio 7

Aqui foi utilizado alguns recursos de shell scripting. O arquivo necessário é chamado jruby.jar, e está gravado em algum lugar abaixo do diretório home do usuário (que podemos abreviar como ~, no meu caso toda vez que utilizo ~ é entendido como /home/taq/), então utilizamos find ~ -iname 'jruby.jar' para encontrá-lo.

Como esse comando está contido entre \$(), o seu resultado já é automaticamente inserido no local, deixando a CLASSPATH como o *path* encontrado, o diretório local e o que já havia nela.

⁴⁹<http://ruby-doc.org/core/classes/Array.htmlM000206>

⁵⁰<http://ruby-doc.org/core/classes/String.htmlM001112>