Conhecendo Ruby

Aprenda de forma prática e divertida

Aborda a versão 3.0 da linguagem

Eustáquio Rangel

Sumário

Esse livro foi escrito todo usando LaTeX, retornando de volta às suas raízes de 2004.

Os arquivos de código-fonte utilizados no livro podem ser acessados no repositório do Github.

Ilustração da capa: Aquarela de Ana Carolina Otero Rangel.

Capítulo 1

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", da Editora Brasport, cujas cópias se esgotaram 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, que sem sombra de dúvidas foi o principal catalizador da linguagem, 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, 2.x e agora, as 3.x.

Esse livro vai ser baseado nas versões **3.x**, mas em qualquer versão **2.x** o código é perfeitamente utilizável, *a não ser quando comentado explicitamente que é um recurso de uma versão mais recente*.

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!

Esse material também serve como base para os treinamentos de Ruby on Rails que ministramos na minha empresa, a Bluefish. Se precisar de cursos de Ruby on Rails, seja na nossa empresa, seja in-company, ou precisar de consultoria em projetos que utilizam essa ferramenta, entre em contato conosco através de contato@bluefish.com.br.

Um grande abraço!

Capítulo 2

Ruby

2.1 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 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, substituindo a mais lenta Ruby MRI (Matz's Ruby Interpreter).

Fonte: Wikipedia

2.2 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 e mais moderna e prática.

2.2.1 Ubuntu

Se você está usando o Ubuntu, pode instalá-la com os pacotes nativos do sistema operacional (mas espere um pouco antes de fazer isso, vamos ver outra forma):

```
$ sudo apt-get install ruby<versao>
```

2.2.2 OSX

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

```
$ port install ruby
```

2.2.3 Windows

E até no Windows tem um instalador automático, o RubyInstaller. Mais detalhes para esse tipo de instalação podem ser conferidas no site oficial da linguagem.

Particularmente eu não recomendo utilizar a linguagem no Windows, aí vai de cada um, mas já aviso que vão arrumar sarna para se coçarem. Uma ótima solução para utilizar no Windows é instalar o WSL e instalar a linguagem nele da forma que vamos ver a seguir.

2.2.4 RVM

Vamos instalar Ruby utilizando a RVM - Ruby Version Manager, 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 ao RVM, temos também a rbenv. 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 onde existe um *shell* Unix (por isso ela não está disponível para Windows, nesse caso, verifique a ferramenta pik), sendo necessário apenas abrir um terminal rodando esse shell e executar:

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

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

```
$ ls ~/.rvm
total 92K
.
..
archives
bin
config
environments
examples
gems
gemsets
help
lib
LICENCE
log
patches
README
rubies
scripts
src
tmp
wrappers
também com o diretório de gems:

$ ls ~/.gem
total 28K
.
..
credentials
ruby
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 `~/.bashrc` (ou, dependendo da sua distribuição Linux, no `~/.bash_profile`):

```
$ echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm"' >> ~/.bashrc
```

Talvez na versão que você esteja utilizando isso não seja mais necessário. Mas para confirmar se é necessário ou se a RVM já se encontra corretamente configurada e instalada, podemos executar os seguintes comandos:

```
$ type rvm | head -n1
rvm é uma função

$ rvm -v
rvm 1.29.10-next (master) by Michal Papis, Piotr Kuczynski, Wayne E. Seguin [https://rvm.io]
```

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:

```
$ rvm notes
  Notes for Linux ( DISTRIB_ID=Ubuntu
  DISTRIB_RELEASE=11.04
  DISTRIB_CODENAME=natty
  DISTRIB_DESCRIPTION="Ubuntu 11.04" )

  # NOTE: MRI stands for Matz's Ruby Interpreter (1.8.X, 1.9.X),
  # ree stands for Ruby Enterprise Edition and rbx stands for Rubinius.

  # curl is required.
  # git is required.
  # patch is required (for ree, some ruby head's).

  # If you wish to install rbx and/or any MRI head (eg. 1.9.2-head)
  # then you must install and use rvm 1.8.7 first.
  # If you wish to have the 'pretty colors' again,
  # set 'export rvm_pretty_print_flag=1' in ~/.rvmrc.
  dependencies:

  # For RVM
  rvm: bash curl git

  # For JRuby (if you wish to use it) you will need:
  jruby: aptitude install curl sun-java6-bin sun-java6-jre
  sun-java6-jdk

  # For MRI & ree (if you wish to use it) you will need
  # (depending on what you   # are installing):
  ruby: aptitude install build-essential bison openssl libreadline5
  libreadline-dev curl git zlib1g zlib1g-dev libssl-dev libsqlite3-0
  libsqlite3-dev sqlite3 libxml2-dev
  ruby-head: git subversion autoconf
  # For IronRuby (if you wish to use it) you will need:
  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:

```
$ sudo aptitude install build-essential bison openssl libreadline5
libreadline-dev curl git zlib1g zlib1g-dev libssl-dev libsqlite3-0
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

```
$ 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 mais atual. Para verificar qual é, visitem a página de downloads da linguagem e verifiquem qual é a versão estável, ou executem esse comando no terminal (levando em conta que já esteja instalado o utilitário curl, que é ferramenta essencial hoje em dia):

```
$ curl https://www.ruby-lang.org/pt/downloads/ 2> /dev/null | grep -o "Ruby [0-9].[0-9].[0-9]" | sort
↪  | uniq | tail -n1
Ruby 3.0.0
```

Ali vimos que a última versão reconhecida (no momento em que estou escrevendo esse texto) é a **3.0.0**.

> **Ferramentas extras**
>
> Ali acima eu utilizei alguns recursos de *shell scripting* para automatizar o processo de recuperar a versão mais recente estável da linguagem. Aprender shell scripting é uma coisa muito interessante hoje em dia pois as ferramentas disponíveis em um terminal (que ainda assusta muita gente adepta do "next-next-finish") são imensamente poderosas e podem economizar muito tempo gasto fazendo de outra maneira.
>
> *"Malditos sejam, ó Terra e Mar, pois o Demônio manda a besta com fúria, porque ele sabe que o tempo é curto. Deixe aquele que ousa tentar entender o número da besta pois ele é um número humano. E seu número é ... next next finish!"*

Levando em conta que <versão> é a última versão que encontramos acima, podemos executar no terminal:

```
$ rvm install <versão>
  Installing Ruby from source to:
  /home/taq/.rvm/rubies/ruby-<versão>
  this may take a while depending on your cpu(s)...
  #fetching
  #downloading ruby-<versão>, this may
  take a while depending on your connection...
  ...
```

Após instalado, temos que ativar a versão na RVM e verificar se ficou ok, digitando o comando rvm seguido do número da versão para ativá-la, o que pode ser conferido logo depois com o comando `ruby -v`:

```
$ rvm <versão>

$ ruby -v
  ruby <versão> [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 versão instalada como a padrão do sistema:

```
$ rvm use <versão> --default
  Using /home/taq/.rvm/gems/ruby-<versão>
```

Outra opção para gerenciar qual versão está ativa, é criar um arquivo chamado `.ruby-version`, com o número da versão que queremos ativar:

```
$ echo <versão> > .ruby-version

$ cat .ruby-version
  <versão>
```

Importante notar que essa versão vai ser ativada somente quando navegarmos para o diretório onde o arquivo se encontra. Ou seja, toda vez que utilizamos, por exemplo, o comando cd para irmos para o diretório onde o arquivo se encontra, a versão especificada vai ser ativada.

A partir da versão 2.1, começou a ser empregado o versionamento semântico, onde é adotado o esquema de MAIOR.MENOR.AJUSTE no número da versão, sendo:

- Maior (major) - existem incompatibilidades de API
- Menor (minor) - adicionadas funcionalidades de modo compatível com versões anteriores
- Ajuste (patch) - correções de bugs de modo compatível com versões anteriores

Com tudo instalado e configurado, podemos prosseguir.

Várias versões, vários interpretadores

Convenhamos, não é toda linguagem por aí que permite instalar várias versões e vários interpretadores da linguagem e permitem alterar entre elas de maneira prática. Algumas são tão complicadas em fazer isso que parecem CMB (Coisa da Maria Barbuda). Usando uma ferramenta como a RVM (e similares) permitem uma facilidade em lidar com seus projetos tanto de versões legadas e muito antigas como projetos de ponta utilizando as versões mais recentes e talvez ainda nem liberadas como estáveis.

Capítulo 3

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, 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?".

3.1 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:

```
v = "teste"

v.class
=> String

v = 1

v.class
=> Integer
```

Pudemos ver que a variável [1] `v` pode assumir como valor tanto uma `String` como um número (que nesse caso, é um `Integer` - 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. Vamos criar um arquivo chamado `Estatica.java` (ou peguem do repositório de código-fonte do livro, indicado nas primeiras páginas):

[1] Variáveis são referências para áreas na memória

```java
public class Estatica {
  public static void main(String args[]) {
    String v = "teste";
    System.out.println(v);
    v = 1;
  }
}
```

Código 1: Tipagem estática em Java

Tentando compilar:

```
$ javac Estatica.java

Estatica.java:5: incompatible types
found : int
required: java.lang.String
v = 1;
^
1 error
```

Performance, segurança, etc

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, para garantir que o software esteja de acordo com o que esperamos dele.

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.x, 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.

3.2 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:

```ruby
i = 1
s = 'oi'
puts i + s
=> TypeError: String can't be coerced into Fixnum
```

Enquanto em uma linguagem como PHP, temos tipagem fraca:

```
<?php
$i = 1;
$s = "oi";
print $i + $s;
?>
```

Código 2: Tipagem fraca em PHP

Rodando isso, resulta em um *warning*, mas retorna de alguma forma `1`:

```
$ php tipagem_fraca.php
PHP Warning:  A non-numeric value encountered in tipagem_fraca.php on line 4
1
```

3.3 Tipos básicos

Não temos primitivos em Ruby, somente abstratos, onde todos exibem comportamento de objetos. Temos números inteiros e de ponto flutuante, onde a linguagem divide automaticamente os inteiros em `Fixnums` e `Bignums`, que são diferentes somente pelo tamanho do número, sendo convertidos automaticamente. A partir da versão 2.4 de Ruby, os inteiros foram convertidos em `Integer`, facilitando em um nível mais alto e mantendo o comportamento anterior em um nível mais baixo, se necessário identificar qual o tipo exato. Vamos ver alguns deles agora.

3.3.1 Inteiros

Os inteiros são os números mais convencionais que temos, sendo que a linguagem os separa *internamente* em dois subtipos. Para todos os efeitos, podemos levar em conta que todos são inteiros, sendo separados apenas pelo seu valor, como vamos ver logo a seguir:

```
> v = 1
=> 1
> v.class
=> Integer
```

Fixnums

Aqui vamos ver como é feita a separação interna dos inteiros de forma similar às versões *antes* das versões 2.4.x da linguagem, para efeito didático de ver como que alguns "primitivos"são controlados pela linguagem de forma a não sobrecarregarem o processamento e ainda exibir comportamento de objetos mesmo para números inteiros.

Os `Fixnums` são números inteiros de 62 bits de comprimento (ou 1 `word` do processador menos 2 bits), 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 64 bits, costumeiras hoje em dia, de 62 bits. Para isso vamos abrir o interpretador de comandos do Ruby, o `irb`. Para acioná-lo, abra um emulador de terminal (você está em um ambiente/sistema operacional que tem um, correto?) e digite `irb`, seguido dos seguintes comandos, levando em conta que o caracter > é o *prompt* de comando do `irb`:

```
$ irb

> (2 ** 62).class
=> Integer

> ((2 ** 62) - 1)
=> 4611686018427387903

> ((2 ** 62) - 1).object_id & 0x1
=> 1
```

Dica

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

Dica

A representação de código sendo executado no `irb` vai ser a seguinte:

`>` prompt de comando, onde o código vai ser digitado e executado apertando `enter`
`=>` resultado do código executado

Quando for encontrado um cifrão (`$`), significa que o código deve ser rodado no `terminal do sistema operacional`, como no exemplo:

```
$ ruby teste.rb
```

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:

```
> n = 1234
=> 1234

> n.object_id
=> 2469

> n.object_id >> 1
=> 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:

```
> n1 = 1234
=> 1234

> n2 = 1234
=> 1234

> n1.object_id
=> 2469

> n2.object_id
=> 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. Se o resultado for `1`, a referência aponta para um tipo de dado que é um `immediate value`. Foi isso que fizemos no primeiro exemplo falando sobre os `Integers` e que vamos fazer

novamente aqui:

```
> n = 1234
=> 1234

> n.object_id & 0x1
=> 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.methods

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

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

3.3.2 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, em um computador de 64 bits:

```
> (2 ** 62)
 => 4611686018427387904
> (2 ** 62).object_id & 0x1
 => 0
```

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 `Bignums` e vendo que cada uma tem um `object_id` diferente:

```
> b1 = (2 ** 62)
=> 4611686018427387904
> b2 = (2 ** 62)
```

```
=> 4611686018427387904
> b1.object_id
=> 320
> b2.object_id
=> 340
```

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

```
> 1_234_567
=> 1234567

> 1_234_567_890
=> 1234567890
```

Diferenciando Fixnums e Bignums a partir das versões 2.4.x

Só para fixar: como vimos anteriormente, somente os `Fixnums` que são `immediate values`, então isso nos permite diferenciar dois `Integers` a partir do `and` lógico do `object_id` (rodando em uma máquina de 64 bits):

```
> ((2 ** 62) - 1).object_id & 0x1
=> 1
> ((2 ** 62) - 0).object_id & 0x1
=> 0
```

Como podemos ver, somente o primeiro que retornou 1, indicando que a representação interna do `Integer` é um `Fixnum`.

Dica

Sempre prefiram um computador com 64 bits. Agora que sabemos como funciona as internas da linguagem, temos noção que são criados bem menos `Bignums` quando a `word` do processador é maior. Tudo bem que alguns valores desses só apareceriam se a sua aplicação estivesse analisando alguns números da "Operação Lava-Jato"e algumas outras parecidas, mas vocês entenderam a idéia. E nem sei se ainda conseguimos encontrar computadores com 32 bits por aí, mas fica a dica!

3.3.3 Ponto flutuante

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

```
> 1.23
=> 1.23

> 1.234
=> 1.234
```

Importante notar que os `Floats` **podem ou não ser** `immediate values`:

```
> f1 = 1.23
=> 1.23
> f2 = 4.2e100
=> 4.2e100
> f3 = 1.23
=> 1.23
```

```
> f4 = 4.2e100
=> 4.2e100
> f1.object_id
=> -27742173704602254
> f2.object_id
=> 260
> f3.object_id
=> -27742173704602254
> f4.object_id
=> 280
```

3.3.4 Racionais

É nóis, mano.

> Mando um recado lá pro meu irmão: Se tiver usando droga, tá ruim na minha mão. Ele ainda tá com aquela mina. Pode crer, moleque é gente fina.

Podemos criar racionais (não os M.C.s) utilizando explicitamente a classe `Rational`:

```
> Rational(1,3)
=> (1/3)

> Rational(1,3) * 9
=> (3/1)
```

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

```
> '1/3'.to_r * 9
=> (3/1)
```

Dica

Podemos converter um número racional para inteiro:

```
> (1/3.to_r * 10).to_i
=> 3
```

Ou ponto flutuante, o que nesse caso, faz mais sentido:

```
> (1/3.to_r * 10).to_f
=> 3.3333333333333335
```

Se quisermos arredondar:

```
> (1/3.to_r * 10).to_f.round(2)
=> 3.33
```

3.3.5 Booleanos

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

```
> true.object_id
=> 2
```

```
> false.object_id
=> 0
```

3.3.6 Nulos

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

```
> nil.object_id
=> 4
```

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

```
> v = 1
=> 1

> v.nil?
=> false

> v = nil
=> nil

> v.nil?
=> true
```

3.3.7 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 \:

```
puts "o céu é 'azul'"
puts "o céu é \"azul\""
puts 'o céu é "azul"'
puts 'o céu é \'azul\''
```

Código 3: Escapando strings

Resultado:

```
o céu é 'azul'
o céu é "azul"
o céu é "azul"
o céu é 'azul'
```

Também podemos criar `Strings` longas, com várias linhas, usando o conceito de `heredoc`, onde indicamos um terminador logo após o sinal de atribuição (igual) e dois sinais de menor («):

```
str = <<FIM
Criando uma String longa
com saltos de linha e
vai terminar logo abaixo.
FIM

puts str
```

Código 4: Heredocs

Resultado:

```
Criando uma String longa
com saltos de linha e
vai terminar logo abaixo.
```

O terminador tem que vir logo no começo da linha onde termina a `String`, e ser o mesmo indicado no começo. Nesse exemplo, foi utilizado o terminador `FIM`. Utilizar `heredocs` evita que façamos muitas linhas cheias de `Strings` uma concatenando com a outra.

Assim como o terminador sempre está no começo da linha, estão as linhas seguintes. Dá usar usar também essa sintaxe para mover o terminador:

```
str = <<-FIM
Criando uma String longa
com saltos de linha e
vai terminar logo abaixo.
        FIM

puts str
```

Código 5: Heredocs com espaços no terminador

Aqui movemos o **terminador** para outra posição fora do começo da linha (isso vai ser útil para usar em métodos), mas não resolve o problema de ter que alinhar todas as frases no começo da linha:

```
str = <<-FIM
      Criando uma String longa
      com saltos de linha e
      vai terminar logo abaixo.
        FIM
puts str
```

Código 6: Heredocs com espaços no terminador e linhas

Resultado:

```
      Criando uma String longa
      com saltos de linha e
      vai terminar logo abaixo.
```

Para remover os espaços no começo da linha e nos permitir alinhar de forma consistente no código, podemos utilizar o operador "squiggly", que nada mais é do que um til (~) antes do terminador:

```
str = <<~FIM
      Criando uma String longa
      com saltos de linha e
      vai terminar logo abaixo.
        FIM

puts str
```

Código 7: Heredocs com squiggly

Resultado:

```
Criando uma String longa
com saltos de linha e
vai terminar logo abaixo.
```

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

```
> s1 = "ola"
=> "ola"
```

```
> s2 = "ola"
=> "ola"

> s1.object_id
=> 260

> s2.object_id
=> 280
```

Mas aqui tem um porém importante em termos de gerenciamento de memória, o que afeta diretamente a parte de performance. Se ao invés de abrirmos o `irb` da forma costumeira, utilizarmos essa forma:

```
$ RUBYOPT=--enable-frozen-string-literal irb
```

E executarmos o código acima novamente:

```
> s1 = "ola"
=> "ola"

> s2 = "ola"
=> "ola"

> s1.object_id
=> 260

> s2.object_id
=> 260
```

Aqui entra um conceito de objetos **congelados** que vamos ver mais para frente do livro, onde vamos ter mais explicações sobre isso. Por enquanto, vamos só guardar o fato de que ambas as `Strings` que são idênticas compartilham o mesmo `object_id`.

3.3.8 Substrings

`Substrings` são partes de uma `String` (antes eu havia escrito "pedaços"de uma `String`, mas ficou um lance muito `Tokyo Ghoul/Hannibal Lecter`, então achei "partes"mais bonito). Para pegar algumas `substrings`, podemos tratar a `String` como um `Array`:

```
> str = "string"
=> "string"

> str[0..2]
=> "str"

> str[3..4]
=> "in"

> str[4..5]
=> "ng"
```

Podendo também usar índices negativos para recuperar as posições relativas ao final da `String`:

```
> str[-4..3]
=> "ri"

> str[-5..2]
=> "tr"
```

```
> str[-4..-3]
=> "ri"

> str[-3..-1]
=> "ing"

> str[-3..]
=> "ing"

> str[-1]
=> "g"

> str[-2]
=> "n"
```

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

```
> str.slice(0,2)
=> "st"

> str.slice(3,2)
=> "in"
```

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

```
# Ruby 1.8.x
> str[0]
=> 115

# Ruby 1.9.x e maiores
> str[0]
=> "s"
```

Mais sobre *encodings*, logo abaixo.

3.3.9 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 `«`:

```
> nome = "Eustaquio"
=> "Eustaquio"

> sobrenome = "Rangel"
=> "Rangel"

> nome + " " + sobrenome
=> "Eustaquio Rangel"

> nome.object_id
=> 84406670

> nome << " "
=> "Eustaquio "

> nome << sobrenome
=> "Eustaquio Rangel"
```

```
> nome.object_id
=> 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, sem gerar um novo objeto.

3.3.10 Encodings

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:
>
> ```
> # 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. Se for outro *encoding*, é só inserir o comentário no arquivo.

Podemos verificar o *encoding* de uma `String`:

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

Podemos definir o *encoding* de uma `String`:

```
> "eustáquio".encode "iso-8859-1"
=> "eust\xE1quio"

> "eustáquio".encode("iso-8859-1").encoding
=> #<Encoding:ISO-8859-1>
```

> Repararam que não precisamos dos parenteses para chamar um método em Ruby, como no primeiro caso de `encode` acima, e utilizando no segundo? Vamos ver isso quando chegarmos na parte de métodos.

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

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

3.3.11 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:

```
> nick = "TaQ"
=> "TaQ"
```

```
> other_nick = nick
=> "TaQ"

> nick[0] = "S"
=> "S"

> other_nick
=> "SaQ"
```

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

```
> nick = "TaQ"
=> "TaQ"

> other_nick = nick.dup
=> "TaQ"

> nick[0] = "S"
=> "S"

> nick
=> "SaQ"

> other_nick
=> "TaQ"
```

3.3.12 Congelando objetos explicitamente

Se, por acaso quisermos de forma explícita que um objeto não seja modificado, podemos utilizar o método `freeze`, que vai disparar uma exceção do tipo `FrozenError` se tentarmos alterar o objeto de alguma forma:

```
> nick = "TaQ"
=> "TaQ"

> nick.freeze
=> "TaQ"

> nick[0] = "S"
FrozenError (can't modify frozen String: "TaQ")
```

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

```
> nick = "TaQ"
=> "TaQ"

> nick.freeze
=> "TaQ"

> new_nick = nick.dup
=> "TaQ"

> new_nick[0] = "S"
=> "SaQ"
```

Importante notar que é criada uma tabela de `Strings` congeladas, assim toda `String` congelada vai ser o mesmo ob-

jeto:

```
> s1 = "taq".freeze
=> "taq"

> s1.object_id
=> 10988480

> s2 = "taq".freeze
=> "taq"

> s2.object_id
=> 10988480
```

Lembram que foi comentado a questão de performance em objetos congelados? Pensem assim, agora ao invés de 2 `Strings`, temos somente uma, que vai atender às nossas necessidades ali nas duas variáveis. Além de dar mais velocidade para o gerenciamento de memória, vai nos dar mais segurança em transmitir essas referências entre métodos, sem efeitos colaterais muitas vezes indesejados. Quando começarmos a utilizar arquivos de código, vamos ver como utilizar um *magic comment* para ativar as `Strings` congeladas no arquivo inteiro.

Os objetos congelados também são chamados de "objetos imutáveis". Temos esse comportamento em algumas outras linguagens e esse comportamento é valorizado especialmente em linguagens funcionais, de onde não são exclusividade. Vamos ver um exemplo de `String` congelada/imutável em Java:

```
public class Imutavel {
   public static void main(String args[]) {
      String mutable = "conteúdo original";
      mutable = "essa string é mutável!";

      final String immutable = "conteúdo original";
      immutable = "essa string é imutável!";
   }
}
```

Código 8: Objetos imutáveis em Java

Tentando compilar:

```
$ javac Imutavel.java
Imutavel.java:7: error: cannot assign a value to final variable immutable
      immutable = "essa string é imutável!";
      ^
1 error
```

Podemos definir **todas** as `Strings` de um arquivo com código fonte como congeladas utilizando o "comentário mágico" `frozen_string_literal: true` no começo do arquivo:

```
# frozen_string_literal: true
```

Vamos ver um *benchmark* (mais sobre isso mais para frente) verificando as diferenças entre utilizar objetos congelados ou não, nesse caso, `Strings`:

```
require 'benchmark'

Benchmark.bm do |bm|
  bm.report('alocando strings') do
    1_000_000.times { s = "alocando!" }
  end
end
```

Código 9: Benchmark com Strings e comentários mágicos

Rodando o código, desabilitando as `Strings` congeladas:

```
 RUBYOPT=--disable-frozen-string-literal ruby code/basico/string_bench.rb
                      user     system      total        real
alocando strings  0.071635   0.000047   0.071682 (  0.071916)
```

E agora, com as `Strings` congeladas:

```
 RUBYOPT=--enable-frozen-string-literal ruby code/basico/string_bench.rb
                     user     system      total        real
alocando strings  0.040436   0.000000   0.040436 (  0.040670)
```

Ali já caiu o tempo em 44%!

3.3.13 Alguns métodos e truques com Strings

```
> str = "tente"

> str["nt"] = "st"
=> "teste"

> str.size
=> 5

> str.upcase
=> "TESTE"

> str.upcase.downcase
=> "teste"

> str.sub("t", "d")
=> "deste"

> str.gsub("t", "d")
=> "desde"

> str.capitalize
=> "Desde"

> str.reverse
=> "etset"

> str.split("t")
=> ["","es","e"]

> str.scan("t")
=> ["t","t"]

> str.scan(/^t/)
=> ["t"]

> str.scan(/./)
=> ["t","e","s","t","e"]
```

Alguns métodos acima, como `sub`, `gsub` e `scan` aceitam expressões regulares (vamos falar delas daqui a pouco) e permitem fazer algumas substituições como essa:

```
> "apenas um [teste]".gsub(/[\[\]]/, { "[" => "(", "]" => ")" })
```

```
=> "apenas um (teste)"
```

Particularmente eu prefiro utilizar expressões regulares para fazer manipulações no começo e fim das `Strings`, mas se quisermos remover os prefixos e sufixos temos `delete_prefix` e `delete_suffix`, demonstrados aqui com o `sub` logo após:

```
> "aprovado".delete_prefix("a")
 => "provado"

> "aprovado".delete_suffix("do")
 => "aprova"

> "aprovado".sub(/^a/, "")
 => "provado"

> "aprovado".sub(/do$/, "")
 => "aprova"
```

3.3.14 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:

```
> :teste.class
=> Symbol

> :teste.object_id
=> 263928

> :teste.object_id
=> 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! Lógico que também podemos minimizar a alocação de `Strings` utilizando sempre o comentário mágico de `Strings` congeladas.

3.3.15 Expressões regulares

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

```
> regex1 = /^[0-9]/
=> /^[0-9]/
```

```
> regex2 = Regexp.new("^[0-9]")
=> /^[0-9]/

> regex3 = %r{^[0-9]}
=> /^[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 teste" =~ regex1
=> 0

> "1 teste" =~ regex2
=> 0

> "1 teste" =~ regex3
=> 0

> "outro teste" !~ regex1
=> true

> "outro teste" !~ regex2
=> true

> "outro teste" !~ regex3
=> true

> "1 teste" !~ regex1
=> false

> "1 teste" !~ regex2
=> false

> "1 teste" !~ regex3
=> false
```

No caso das expressões que "casaram", foi retornada a posição da `String` onde houve correspondência. Também podemos utilizar, a partir da versão `2.4`, o método `match?`:

```
> regex1.match? "1 teste"
=> true

> regex1.match? "outro teste"
=> false
```

O detalhe é que o método `match?`, que retorna somente um *boolean* indicando se a expressão "casou"ou não (diferente do `=~` que retorna **onde** casou) é até **3 vezes mais rápido**! Então é preferível utilizar ele se você não precisa saber onde foi o ponto exato que a expressão "casou", somente se ela "casou"ou não.

Podemos fazer truques bem legais com expressões regulares e `Strings`, como por exemplo, dividir a nossa `String` através de uma expressão regular, encontrando todas as palavras que começam com `r`:

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

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

Outra dica interessante é o construtor %r, mostrado acima. Quando temos barras para "escapar"dentro da expressão regular, ele nos permite economizar alguns caracteres, como nesse exemplo:

```
> /\Ahttp:\/\/(www\.)?eustaquiorangel\.com\z/.match? "http://eustaquiorangel.com"
=> true

> %r{http://(www\.)?eustaquiorangel\.com}.match? "http://eustaquiorangel.com"
=> true
```

Também podemos utilizar interpolação de expressão dentro da expressão regular:

```
> host = "eustaquiorangel.com"
=> "eustaquiorangel.com"

> %r{\Ahttp://(www\.)?#{host}\z}.match? "http://eustaquiorangel.com"
=> true
```

Dica

Reparem que eu usei os marcadores de início de linha \A e de fim de linha \z. Geralmente costumamos utilizar os caracteres circunflexo e cifrão, mas tem um pequeno grande problema: esses marcadores não levam em conta a quebra de linha (\n) dentro da String:

```
> /^\w+$/.match?("hello world")
=> false

> /^\w+$/.match?("hello\nworld")
=> true
```

Enquanto que utilizando \A e \z, funciona perfeitamente:

```
> /\A\w+\z/.match?("hello\nworld")
=> false
```

Grupos

Podemos utilizar grupos nas expressões regulares, utilizando (e) para delimitar o grupo, e $<número> para verificar onde o grupo "casou". Não é uma prática muito utilizada, fica mais legível até utilizar grupos nomeados (que já vamos ver) mas está aí se precisar:

```
> "Alberto Roberto" =~ /(\w+)( )(\w+)/
=> 0

> $1
=> "Alberto"

> $2
=> " "

> $3
=> "Roberto"
```

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

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

A partir das versões 2.x, o **Onigmo** é 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:

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

No código acima pudemos ver que foi utilizado o método `p`, que é similar ao método `inspect`, que inspeciona o conteúdo de uma variável sem tentar converter em texto, como uma `String`, quando utilizamos o método `puts`. Vamos dar uma olhada nos jeitos de inspecionar aquela variável `regexp` e notar as diferenças:

```
> p regexp
/^([A-Z])?[a-z]+(?(1)[A-Z]|[a-z])$/
 => /^([A-Z])?[a-z]+(?(1)[A-Z]|[a-z])$/

> puts regexp
(?-mix:^([A-Z])?[a-z]+(?(1)[A-Z]|[a-z])$)
 => nil

> regexp.inspect
 => "/^([A-Z])?[a-z]+(?(1)[A-Z]|[a-z])$/"
```

Como regra, se quisermos a visualização do objeto como texto (o que às vezes não nos dá muita informação), podemos utilizar `puts`. Quando quisermos ver as internas do objeto, utilizamos `p`, se quisermos armazenar essa descrição de forma textual, utilizamos `inspect`.

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

Melhor do que utilizar os marcadores mostrados acima, podemos utilizar **grupos nomeados** em nossas expressões regulares, como por exemplo:

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

> matcher[:objeto]
=> "roupa"

> matcher[:cidade]
=> "Roma"
```

O formato de captura é `(?<nome_do_grupo>)`, onde `nome_do_grupo` vai ser a chave da `Hash` (já vamos ver esse tipo de objeto), convertida em um símbolo, e vai conter tudo o que a expressão regular que estiver dentro dos parenteses "casar".

A partir da versão `2.4`, temos o método `named_captures` que nos retorna uma `Hash` com os valores que foram capturados:

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

> matcher.named_captures
=> {"objeto"=>"roupa", "cidade"=>"Roma"}
```

Caracteres acentuados

E se precisarmos utilizar caracteres com acento nas expressões? Por exemplo, eu juro que o meu nome está correto, mas:

```
> "eustáquio".match? /\A\w+\z/
=> false
```

Para resolver esse problema, podemos utilizar tanto as propriedades de caracteres[2]:

```
> "eustáquio".match? /\A\p{Latin}+\z/
=> true
```

como a indicação de que os caracteres são `Unicode`:

```
> "eustáquio".match? /\A(?u)\w+\z/
=> true
```

3.3.16 Arrays

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

```
> 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`:

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

Dica

Para criarmos `Arrays` de `Strings` o método convencional é:

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

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

e também podemos utilizar `%i` para criar um `Array` de símbolos:

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

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

[2] http://ruby-doc.org/core-2.1.5/Regexp.html#class-Regexp-label-Character+Properties

```
> array = Array.new(5)
=> [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:

```
> array = Array.new(5, 0)
=> [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:

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

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

```
> array[0].upcase!
=> "OI"

> array
=> ["OI", "OI", "OI", "OI", "OI"]
```

> **Dica**
>
> Se tentarmos fazer esse exemplo com a opção de `Strings` congeladas, esse exemplo **não vai funcionar**!

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`:

```
> array = Array.new(5) { "oi" }
=> ["oi", "oi", "oi", "oi", "oi"]

> array[0].upcase!
=> "OI"

> array
=> ["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:

```
> array.each { |numero| puts "O Array tem o numero " + numero.to_s }

=> O Array tem o numero 1
```

```
=> O Array tem o numero 2
=> O Array tem o numero 3
=> 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>|` (o caracter `|` é o *pipe* no Linux). 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`:

```
> a = %w(john paul george ringo)
=> ["john", "paul", "george", "ringo"]

> a[0..1]
=> ["john", "paul"]

> a[1..2]
=> ["paul", "george"]

> a[1..3]
=> ["paul", "george", "ringo"]

> a[0]
=> "john"

> a[-1]
=> "ringo"

> a.first
=> "john"

> a.last
=> "ringo"

> 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:

```
> a = %w(john paul george ringo)
```

```
> a.push("stu")
=> ["john", "paul", "george", "ringo", "stu"]

> a << "george martin"
=> ["john", "paul", "george", "ringo", "stu", "george martin"]
```

Para adicionar elementos no começo de um `Array`, podemos utilizar `unshift`:

```
> a = %w(john paul george ringo)
 => ["john", "paul", "george", "ringo"]

> a.unshift("stu")
 => ["stu", "john", "paul", "george", "ringo"]
```

Se preferirmos, podemos utilizar os *aliases* de `unshift` e `push`, `prepend` e `append`:

```
> a = %w(john paul george)
 => ["john", "paul", "george"]

> a.prepend("stu")
 => ["stu", "john", "paul", "george"]

> a.append("ringo")
 => ["stu", "john", "paul", "george", "ringo"]
```

Se quisermos criar um novo `Array` a partir de outros, deixando seus elementos únicos, podemos utilizar `union`:

```
> ["a", "b", "c"].union ["c", "d", "a"]
 => ["a", "b", "c", "d"]
```

Esse método só definido na versão 2.6 da linguagem, mas antes podíamos (e ainda podemos) utilizar o operador `bitwise OR |` para o mesmo comportamento:

```
> ["a", "b", "c"] | ["c", "d", "a"]
 => ["a", "b", "c", "d"]
```

Para saber os elementos que são diferentes entre um `Array` e outro, também na versão 2.6 foi introduzido o método `difference`, que também era utilizado o operador `-` para a mesma finalidade:

```
> ["a", "a", "b", "b", "c", "c", "d", "e"].difference ["a", "b", "d"]
 => ["c", "c", "e"]

> ["a", "a", "b", "b", "c", "c", "d", "e"] - ["a", "b", "d"]
 => ["c", "c", "e"]
```

E para completar, saber os elementos que são comuns entre um `Array` e outro, na versão 2.7 foi introduzido o método `intersection`, que também era utilizado o operador `bitwise AND &` para a mesma finalidade:

```
> ["a", "b", "c", "d"].intersection ["b", "d"]
 => ["b", "d"]

> ["a", "b", "c", "d"] & ["b", "d"]
 => ["b", "d"]
```

Se quisermos pesquisar em `Arrays` dentro de `Arrays`, podemos utilizar o método `dig`:

```
> array = [0, [1, [2, 3]]]
=> [0, [1, [2, 3]]]
```

```
> array[1][1][0]
=> 2

> array.dig(1, 1, 0)
=> 2
```

3.3.17 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?`:

```
> String.new.respond_to?(:<<)
=> true

> Array.new.respond_to?(:<<)
=> 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?`:

```
> String.new.kind_of?(String)
=> true

> 1.kind_of?(Fixnum)
=> true

> 1.kind_of?(Numeric)
=> true

> 1.kind_of?(Bignum)
=> false
```

3.3.18 Ranges

`Ranges` são intervalos em que existe a *intenção* de criar um intervalo, não a sua *instanciação*, ou seja, definimos os limites, mas não criamos os objetos dentro desses limites. Nos limites podemos definir incluindo ou não o último valor referenciado. Vamos exemplificar isso com o uso de iteradores, dessa maneira:

```
> range1 = (0..10)
=> 0..10

> range2 = (0...10)
=> 0...10

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

```
=> 0..10
> range2.each { |valor| print "#{valor} " }

=> 0 1 2 3 4 5 6 7 8 9
=> 0...10
```

Como pudemos 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 da 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`:

```
> ("a".."z").each { |valor| print "#{valor} " }
=> 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

=> "a".."z"
> ("ab".."az").each { |valor| print "#{valor} " }

=> 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
=> "ab".."az"
```

Outro bem legal é converter uma `Range` em um `Array`, mas atenção: aqui saímos da *intenção* de ter os objetos para a *instanciação* dos objetos, ou seja, não tentem fazer isso com um intervalo de milhões de objetos:

```
> ("a".."z").to_a
=> ["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"]
```

As `Ranges` também podem ser infinitas, com um valor inicial e sem o valor final:

```
> range = (10..)
 => 10..

> range.take(5)
 => [10, 11, 12, 13, 14]
```

Novamente, fica o conselho de não tentar fazer a instanciação de todos os objetos para um `Array`, até porque para o caso de `Ranges` infinitas isso irá gerar uma exceção do tipo `RangeError`.

3.3.19 Hashes

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

```
> hash = { :john => "guitarra e voz", :paul => "baixo e voz", :george => "guitarra", :ringo => "bateria" }
=> {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra", :ringo=>"bateria"}
```

Alerta de coisa antiga

A partir de Ruby 1.9.x as `Hashes` mantém 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, que no caso acima, são símbolos:

```
> hash[:paul]
=> "baixo e voz"

> hash[:ringo]
=> "bateria"
```

Utilizar símbolos como chaves de `Hashes` é uma operação costumeira em Ruby. Se utilizarmos `Strings`, elas serão tratadas como Strings congeladas, o que podemos verificar comparando os seus `object_ids`:

```
> h1 = { "name" => "John" }
=> {"name"=>"John"}

> h2 = { "name" => "Paul" }
=> {"name"=>"Paul"}

> h1.keys.first.object_id
=> 12256640

> h2.keys.first.object_id
=> 12256640
```

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

```
> hash = { "fixnum" => 1, :float => 1.23, 1 => "um" }
=> {1=>"um", :float=>1.23, "fixnum"=>1}

> hash["fixnum"]
=> 1

> hash[:float]
=> 1.23

> hash[1]
=> "um"
```

Podemos criar `Hashes` com valores default:

```
> hash = Hash.new(0)
=> {}

> hash[:um]
=> 0

> hash[:dois]
=> 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:

```
> hash = Hash.new(Time.now)
=> {}
```

```
> hash[:um]
=> Tue Jun 05 23:53:22 -0300 2011

> hash[:dois]
=> 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`:

```
> hash = Hash.new { Time.now }
=> {}

> hash[:um]
=> 2008-12-31 11:31:28 -0200

> hash[:dois]
=> 2008-12-31 11:31:32 -0200

> hash[:tres]
=> 2008-12-31 11:31:36 -0200
```

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

Como utilizamos **muito** símbolos como chaves de `Hashes`, podemos criar dessa maneira:

```
> hash = {john: "guitarra e voz", paul: "baixo e voz", george: "guitarra",
ringo: "bateria"}
=> {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra",
:ringo=>"bateria"}
```

Reparem que o operador "rocket"(=>) sumiu e por convenção esse tem sido o método utilizado desde então. Se você quer ser *vintage*, continue usando o "rocket", se quiser ser *cool*, adote a forma nova. E fique de olho quando for *vintage* te transformar automaticamente em *cool*, tem uma galerinha aí cheia dessas firulas ...

Se desejarmos transformar as chaves de uma `Hash`, podemos utilizar o método `transform_keys`:

```
> hash = { john: "guitarra e voz", paul: "baixo e voz", george: "guitarra",
ringo: "bateria" }
 => {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra",
 :ringo=>"bateria"}

> hash.transform_keys { |key| key.to_s.upcase }
 => {"JOHN"=>"guitarra e voz", "PAUL"=>"baixo e voz", "GEORGE"=>"guitarra",
 "RINGO"=>"bateria"}
```

Se desejarmos **outra** `Hash` apenas com as chaves selecionadas, podemos utilizar `slice`, que mantém a original intocada:

```
> hash = { john: "guitarra e voz", paul: "baixo e voz", george: "guitarra",
ringo: "bateria" }
 => {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra", :ringo=>"bateria"}

> hash.slice(:john, :paul)
 => {:john=>"guitarra e voz", :paul=>"baixo e voz"}

> hash
```

```
 => {:john=>"guitarra e voz", :paul=>"baixo e voz", :george=>"guitarra",
 :ringo=>"bateria"}
```

Podemos utilizar, assim como em `Arrays`, o método `dig` para "cavar"nas estruturas das `Hashes`:

```
> hash = { a: { b: { c: 3 } } }
=> {:a=>{:b=>{:c=>3}}}

> hash[:a][:b][:c]
=> 3

> hash.dig(:a, :b, :c)
=> 3
```

3.3.20 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 agradar. Como exemplo de blocos, temos:

```
> { "Oi, mundo" }
```

e

```
> do
>     puts "Oi, mundo"
>     puts "Aqui tem mais linhas!"
> 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).

Blocos são um recurso muito importante em Ruby e vamos ver mais sobre o seu uso daqui a pouco.

3.3.21 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:

```
# Fixnum para Float
> 1.to_f
=> 1.0

# Fixnum para String
> 1.to_s
=> "1"

# String para Fixnum
> "1".to_i
=> 1

# String inválida para Fixnum
> "a".to_i
=> 0
```

```
# String inválida para Integer, dando erro
> Integer("a")
=> ArgumentError (invalid value for Integer(): "a")

# String para flutuante
> "1".to_f
=> 1.0

# String para símbolo
> "azul".to_sym
=> :azul

# Array para String
> [1, 2, 3, 4, 5].to_s
=> "[1, 2, 3, 4, 5]"

# Array para String, com delimitador
> [1, 2, 3, 4, 5].join(",")
=> "1,2,3,4,5"

# Range para Array
> (0..10).to_a
=> [0,1,2,3,4,5,6,7,8,9,10]

# Hash para Array
> { john: "guitarra e voz" }.to_a
=> [ [:john, "guitarra e voz"] ]
```

3.3.22 Conversões de bases

De inteiro para binário:

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

De binário para decimal:

```
> "10".to_i(2)
=> 2

> 0b10.to_i
=> 2
```

De inteiro para hexadecimal:

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

De hexadecimal para inteiro:

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

3.3.23 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:

```
begin
  numero = 1
  string = "oi"
  numero + string
rescue StandardError => exception
  puts "Ocorreu um erro: #{exception}"
end
```

Código 10: Tratando exceções

Rodando o programa, temos:

```
$ ruby exc.rb
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:

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

Código 11: Tratando exceções com tipo especificado

Rodando o programa, temos:

```
$ ruby exc2.rb
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`:

```
begin
  numero = 1
  string = "oi"
  numero + string
rescue => exception
  puts "Ocorreu um erro do tipo #{exception.class}: #{exception}"
ensure
  puts "Lascou tudo."
end
puts "Fim de programa."
```

Código 12: Tratando exceções com garantia de execução

Rodando o programa:

```
$ ruby exc3.rb
Ocorreu um erro do tipo TypeError: String can't be coerced into Fixnum
Lascou tudo.
Fim de programa.
```

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

```
begin
  numero = 1
  string = "oi"
  numero + string
rescue => exception
  puts "Ocorreu um erro do tipo #{exception.class}: #{exception}"
  puts msg
ensure
  puts "Lascou tudo."
end
puts "Fim de programa."
```

Código 13: Tratando exceções com garantia de execução - mesmo

Rodando o programa:

```
$ ruby exc4.rb
Ocorreu um erro do tipo TypeError: String can't be coerced into Fixnum
Lascou tudo.
exc4.rb:7: undefined local variable or method `msg' for main:Object (NameError)
```

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`:

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

Código 14: Tratando exceções com retry

Rodando o programa:

```
$ ruby retry.rb
Ops, problemas aqui (TypeError), vou tentar de novo.
3
```

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

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

Código 15: Tratando exceções com backtrace

Rodando o programa:

```
$ ruby backtrace.rb
Ops, problemas aqui (TypeError), vou tentar de novo.
backtrace.rb:4:in `+'
backtrace.rb:4
3
```

Disparando exceções

Podemos disparar exceções utilizando `raise`:

```
numero1 = 1
numero2 = 1

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

Código 16: Disparando exceções com raise

Descobrindo 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):

```
begin
  begin
    raise 'foo'
  rescue Exception => foo
    raise 'bar'
  end
rescue Exception => bar
  puts "a causa de #{bar} foi #{bar.cause}"
end
```

Código 17: Descobrindo a exceção anterior

Para versões anteriores, dá para utilizar ou a `gem cause` [3].

Criando nossas próprias exceções

[3] https://github.com/ConradIrwin/cause

Alerta de carro na frente dos bois

Vamos ver aqui um conceito de herança que vamos nos aprofundar somente quando começarmos a ver como que definimos nossas próprias classes. Para não perder o embalo falando de exceções, vamos fazer esse pequeno desvio do futuro e quando chegarmos na parte de classes vamos ter uma explicação melhor.

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`:

```
class NameNotEqual < StandardError
  def initialize(current, expected)
    super "Você digitou um nome inválido (#{current})! Era esperado #{expected}."
  end
end

begin
  correct = "eustaquio"
  puts "Digite o meu nome: "
  name = gets.chomp

  raise NameNotEqual.new(name, correct) if name != correct

  puts "Digitou correto!"
rescue NameNotEqual => e
  puts e
end
```

Código 18: Criando nossas próprias exceções customizadas

Rodando o programa e digitando qualquer coisa diferente de "eustaquio":

```
$ ruby customexceptions.rb
Digite o meu nome:
barizon
Você digitou um nome inválido (barizon)! Era esperado eustaquio.
```

Comparando exceções

Podemos fazer comparações entre duas exceções, como:

```
> Exception.new == Exception.new
=> true
```

Elas vão ser diferentes se tiverem mensagens diferentes:

```
> Exception.new("hello") == Exception.new("world")
=> false
```

Que funciona com a nossa exceção customizada demonstrada acima:

```
> NameNotEqual.new("eustaquio", "rangel") == NameNotEqual.new("eustaquio", "rangel")
=> true

> NameNotEqual.new("eustaquio", "rangel") == NameNotEqual.new("taq", "rangel")
=> false
```

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`:

```
def get_input
  puts "Digite algo (número termina):"
  resp = gets

  throw :end_of_response, resp if resp.chomp.match? /^\d+$/

  resp
end

num = catch(:end_of_response) do
  while true
    get_input
  end
end

puts "Terminado com: #{num}"
```

Código 19: Tratando exceções com catch e throw

Rodando o programa:

```
$ ruby catchthrow.rb
Digite algo (número termina):
oi
Digite algo (número termina):
123
Terminado com: 123
```

3.4 Estruturas de controle

3.4.1 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`:

```
i = 10
if i == 10
  puts "i igual 10"
else
  puts "i diferente de 10"
end
```

Código 20: If

Rodando o programa:

```
$ ruby if.rb
i igual 10
```

This is the end
Beautiful friend
This is the end
My only friend, the end

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:

```
> puts "i igual 10" if i == 10
=> i igual 10
```

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

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

```
i = 10

if i > 10
  puts "maior que 10"
elsif i == 10
  puts "igual a 10"
else
  puts "menor que 10"
end
```

Código 21: Elseif

Rodando o programa:

```
$ ruby elsif.rb
igual a 10
```

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

```
i = 10

result =
if i > 10
  "maior que 10"
elsif i == 10
  "igual a 10"
else
  "menor que 10"
end

result
```

Código 22: Capturando a saída de um if

Rodando o programa:

```
$ ruby captureif.rb
"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:

```
> nome = nil
> puts "Olá, desconhecido!" unless nome
```

O que aconteceu ali é que imprimimos a mensagem *a não ser* que a variável nome tenha algum conteúdo válido (diferente de `nil` e `false`).

Seria basicamente um `if !nome`. A forma de `if` negativo pode dar algum nó na cabeça algumas vezes, mas é bem útil em outras.

case

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

```
i = 10

case i
when 0..5
  puts "entre 0 e 5"
when 6..10
  puts "entre 6 e 10"
else
  puts "hein?"
end
```

Código 23: Case

Rodando o programa:

```
$ ruby case.rb
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:

```
i = 10

case i
when Fixnum
  puts "Número!"
when String
  puts "String!"
else
  puts "hein???"
end
```

Código 24: Comparando tipos com case

Rodando o programa:

```
$ ruby casetype.rb
Número!
```

Para provar que esse teste tem precedência, podemos fazer:

```
i = 10

case i
when Fixnum
  puts "Número!"
when (0..100)
  puts "entre 0 e 100"
end
```

Código 25: Precedência em case

Rodando o programa:

```
$ ruby caseprec.rb
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:

```
> Fixnum === 1
=> true

> Fixnum === 1.23
=> false
```

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

```
> /[0-9]/ === "123"
=> true

> /[0-9]/ === "abc"
=> false
```

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

```
> (0..10) === 1
=> true

> (0..10) === 100
=> false
```

Splat

Convém dedicarmos um tempo para explicar um operador bem útil em Ruby, que é o operador `splat` (asterisco, `*`)

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:

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

> p *array
=> "um"
=> "dois"
=> "tres"
=> nil

> hash = { :um => 1, :dois => 2, :tres => 3 }
=> { :um => 1, :dois => 2, :tres => 3 }

> p *hash
=> [:tres, 3]
=> [:um, 1]
=> [:dois, 2]
=> nil
```

E quando utilizamos antes de um nome de uma variável, ele "suga"os valores excedentes, agindo como um "buraco negro". Testando com uma atribuição simples:

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

> a, b, *c = *array
 => [1, 2, 3, 4]

> a
 => 1

> b
 => 2

> c
 => [3, 4]
```

Pattern matching

O recurso de *pattern matching* foi apresentado por Kazuki Tsujimoto na RubyKaigi 2019, onde foi utilizada uma definição de "Learn You a Haskell for Great Good!", de Miran Lipovaca:

> "A correspondência de padrões (*pattern matching* consiste em especificar padrões aos quais alguns dados devem estar em conformidade e em seguida, verificar se isso ocorre e desconstruir os dados de acordo com esses padrões."

Vamos ver um exemplo utilizando o exemplo de atribuição com o operador `splat` como visto acima. Podemos utilizar uma sintaxe recente para fazer esse tipo de operação, que é a *one line pattern matching*, mas quando esse livro foi atualizado era tão nova que recebemos um *warning*:

```
> [1, 2, 3, 4] => [a, b, *c]
(irb) warning: One-line pattern matching is experimental, and the behavior may
change in future versions of Ruby!
 => nil

> a
 => 1

> b
 => 2

> c
 => [3, 4]
```

Esse tipo de operação pode ser chamada de `desestruturação`. Isso é útil para operações com o `splat`, como por exemplo, tentando utilizar uma `Hash` no lado direito do `splat`:

```
> hash = { a: 1, b: 2, c: 3, d: 4 }
 => {:a=>1, :b=>2, :c=>3, :d=>4}

a, b, *c = *hash
 => [[:a, 1], [:b, 2], [:c, 3], [:d, 4]]

> a
 => [:a, 1]

> b
 => [:b, 2]

3> c
 => [[:c, 3], [:d, 4]]
```

Nesse caso, cada variável ficou com um par de chave e valor da `Hash`. Utilizando o *one-line pattern matching*, podemos fazer:

```
> hash => { a:, b:, **c }
(irb):26: warning: One-line pattern matching is experimental, and the behavior may change in future versions
 => nil

> a
 => 1

> b
 => 2

 > c
 => {:c=>3, :d=>4}
```

Vejam que foram criadas variáveis cujos nomes são as chaves da `Hash` utilizada do lado direito do `pattern matching`. Importante notar que as chaves tem que "casar"com o que foi definido do lado direito, senão vamos ter um erro de *matching*:

```
> hash => { a:, b:, e:}
(irb):45: warning: One-line pattern matching is experimental, and the behavior may change in future versions
NoMatchingPatternError
```

Vamos utilizar o nosso amigo `case` para algo como:

```
valores = [1, 2, 3]

case valores
  in [a]
    puts "O valor é #{a}"
  in [a, b]
    puts "Os valores são: #{a} e #{b}"
  in [a, b, c]
    puts "Os valores são: #{a}, #{b} e #{c}"
end
```

Código 26: Pattern matching com arrays

Rodando o exemplo:

```
Os valores são: 1, 2 e 3
```

Reparem no primeiro `case` que temos 3 elementos `Array`, que combinou com o `in` que tem as três variáveis `a`, `b` e `c`.

Reparem que conseguimos além de fazer o "encaixe", também identificar valores dentro do `Array`:

```
traducoes = [:br, 'Bom dia', :en, 'Good morning']

case traducoes
  in [:br, texto_br, :es, texto_es]
    puts "'#{texto_br}' em Espanhol é '#{texto_es}'"
  in [:br, texto_br, :en, texto_en]
    puts "'#{texto_br}' em Inglês é '#{texto_en}'"
  else
    puts "Sem tradução"
end
```

Código 27: Pattern matching com arrays, identificando elementos

Rodando o exemplo:

```
'Bom dia' em Inglês é 'Good morning'
```

Nesse caso com símbolos e `Strings`, `:br` (no índice 0) e `:en` (no índice 2) combinaram com a expressão da segunda cláusula `in`. Poderíamos ter utilizado `Hashes` também, que vai dar o mesmo resultado anterior:

```
traducoes = { original: :br, texto_original: 'Bom dia', destino: :en, texto_traduzido: 'Good morning' }

case traducoes
  in { original: :br, texto_original: texto_original, destino: :es, texto_traduzido: texto_traduzido }
    puts "'#{texto_original}' em Espanhol é '#{texto_traduzido}'"
  in { original: :br, texto_original: texto_original, destino: :en, texto_traduzido: texto_traduzido }
    puts "'#{texto_original}' em Inglês é '#{texto_traduzido}'"
  else
    puts "Sem tradução"
end
```

Código 28: Pattern matching com Hashes

Podemos utilizar o operador `splat` fazendo com que os resultados sejam concentrados em uma variável:

```
valores = [1, 2, 3]

case valores
  in [a, *b]
    puts "O valor de a é #{a}"
    puts "b vale:"
    p b
end
```

Código 29: Pattern matching com splat

Rodando o exemplo:

```
O valor de a é 1
b vale:
[2, 3]
```

Se por acaso quisermos ignorar um "marcador"do padrão, podemos utilizar o sublinhado (*underscore*) de forma a aproveitar somente o conteúdo que está fora dos sublinhados, no caso o conteúdo vai ser inserido na variável `dois` que está definida.

```
valores = [1, 2, 3]

case valores
  in [_, dois, _]
    puts "O segundo valor é #{dois}"
end
```

Código 30: Pattern matching ignorando posições

Rodando o exemplo:

```
O segundo valor é 2
```

O formato com o *pin operator* faz a avaliação da variável ao invés de atribuir o valor nela. Pegando como exemplo o código abaixo onde `[1, 2, 2]`, `a` é atribuída com `1`, `b` com `2`, e agora como temos `^b`, ao invés do terceiro elemento do `Array` ser atribuído á variável, é verificado se o valor do terceiro elemento do `Array` combina com o valor já atribuído anteriormente à variável, o que procede.

Rodando o exemplo:

```
Os valores com pin são: 1 e 2
```

```
case [1, 2, 2]
  in [a, b, ^b]
    puts "Os valores com pin são: #{a} e #{b}"
end
```

Código 31: Pattern matching com o pin operator

Utilizando a atribuição de seta *arrow assignment*, podemos armazenar em uma variável o padrão que combinou.

```
traducoes = [:br, 'Bom dia', :en, 'Good morning']

case traducoes
  in [Symbol, String, Symbol, String] => found
    p found
end
```

Código 32: Pattern matching com arrow assignment

Rodando o exemplo:

```
[:br, "Bom dia", :en, "Good morning"]
```

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
> ```
>
> Os arquivos com os códigos dos exemplos estão disponíveis em um repositório.

Aqui fica uma dica de que podemos utilizar um "comentário mágico" para policiar os nossos arquivos de código fonte. Ruby vai ler o seu código independente do bom alinhamento do código (não força igual `Python`, mas vamos ser legais e lembrar que tem **pessoas** lendo o seu código, além de computadores executando, beleza? Isso faz parte de uma filosofia/metodologia de *Clean Code*, que eu recomendo fortemente. Vamos ver um exemplo de código desalinhado que funciona:

```
v = 1
if v == 1
  puts 'Valor é igual a 1!'
  end
```

Código 33: Código desalinhado que funciona

Rodando o código:

```
$ ruby not_aligned_working.rb
Valor é igual a 1!
```

Agora vamos inserir o comentário mágico `warn_indent: true` e testar novamente:

```
# warn_indent: true

v = 1
if v == 1
  puts 'Valor é igual a 1!'
  end
```

Código 34: Código desalinhado que funciona

Rodando o código:

```
$ ruby not_aligned_working_warning.rb
not_aligned_working_warning.rb:6: warning: mismatched indentations at 'end' with 'if' at 4
Valor é igual a 1!
```

Pudemos ver que recebemos um alerta sobre a indentação estar incorreta.

while

Faça enquanto:

```
i = 0

while i < 5
  puts i
  i += 1
end
```

Código 35: While

Rodando:

```
$ ruby while.rb
0
1
2
3
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):

```
for i in (0..5)
  puts i
end
```

Código 36: For

Rodando:

```
$ ruby for.rb
0
1
2
3
```

```
4
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`:

```
for i in (0..5)
  break if i == 3
  puts i
end
```

Código 37: For com break

Rodando:

```
$ ruby for.rb
0
1
2
```

Agora o `next`:

```
for i in (0..5)
  next if i == 3
  puts i
end
```

Código 38: For com next

Rodando:

```
$ ruby next.rb
0
1
2
4
5
```

Agora o `redo`:

```
for i in (0..5)
  redo if i == 3
  puts i
end
```

Código 39: For com redo

Rodando:

```
$ ruby for.rb
0
1
2
for.rb:2: Interrupt
from for.rb:1:in ‘each’
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:

```
i = 0

until i==5
  puts i
  i += 1
end
```

Código 40: Until

Rodando:

```
$ ruby until.rb
0
1
2
3
4
```

> **Dica**
>
> Não temos os operadores `++` e `-- em Ruby. Utilize += e -=.`

Operadores lógicos

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

```
> a = 1
=> 1

> b = 2
=> 2

> c = a && b
=> 2

> c
=> 2

> d = a and b
=> 2

> d
=> 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:

```
> a ||= 10
=> 10
```

```
> a
=> 10

> a ||= 20
=> 10

> a
=> 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:

```
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:

```
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:

```
> a, b, c = (0..2).to_a
=> [0, 1, 2]

> defined? a
=> "local-variable"

> defined? b
=> "local-variable"

> defined? String
=> "constant"

> defined? 1.next
=> "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!

Um operador um pouco diferente

Temos em Ruby o operador `flip-flop`, que segue um comportamento parecido com o *flip-flop* que temos em eletrônica, onde é memorizado um estado, onde ele é avaliado como `true` quando avalia a primeira condição como `true` e continua assim até avaliar a segunda condição como `true`, quando passa a retornar `false` até que a primeira condição avalie como `true` novamente.

É um negócio meio complicado de explicar, mas vamos verificar um exemplo dele quando estivermos vendo leitura de arquivos, vai ficar mais fácil de entender.

3.5 Procs e lambdas

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

```
> vezes3 = Proc.new { |valor| valor * 3 }
=> #<Proc:0xb7d959c4@(irb):1>

> vezes3.call(3)
=> 9

> vezes3.call(4)
=> 12

> vezes3.call(5)
=> 15
```

Comportamento similar pode ser alcançado usando `lambda`:

```
> vezes5 = lambda { |valor| valor * 5 }
=> #<Proc:0xb7d791d4@(irb):5>

> vezes5.call(5)
=> 25

> vezes5.call(6)
=> 30

> vezes5.call(7)
=> 35
```

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

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

E também o atalho `.`, menos comum:

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

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

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

> **Dica**
>
> Fica uma dica 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:

```
> pnew = Proc.new { |x, y| puts x + y }
=> #<Proc:0x8fdaf7c@(irb):7>
```

```
> lamb = lambda { |x, y| puts x + y }
=> #<Proc:0x8fd7aac@(irb):8 (lambda)>

> pnew.call(2, 4, 11)
=> nil

> lamb.call(2, 4, 11)
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:

```
def testando_proc
  p = Proc.new { return "Bum!" }
  p.call
  "Nunca imprime isso."
end

puts testando_proc
```

Código 41: Retornando de uma Proc

Rodando:

```
$ ruby procret.rb
Bum!
```

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

```
def testando_lambda
  l = lambda { return "Oi!" }
  l.call
  "Imprima isso."
end

puts testando_lambda
```

Código 42: Retornando de uma lambda

Rodando:

```
$ ruby lambret.rb
Imprima isso.
```

Temos suporte também à sintaxe "stabby proc/lambda", onde podemos utilizar `->` indicando que vamos definir o corpo da `lambda`, opcionalmente indicando quais são seus parâmetros:

```
> p = ->(x, y) { x* y }

> puts p.call(2,3)
=> 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:

```
> mult = lambda { |n1, n2| n1 * n2 }
=> #<Proc:0x8fef1fc@(irb):13 (lambda)>

> mult.(2, 3)
=> 6
```

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

```
> mult = lambda { |n1, n2| n1 * n2 }.curry
=> #<Proc:0x8ffe4e0 (lambda)>

> mult.(2).(3)
=> 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:

```
> mult = lambda { |x| lambda { |y| x * y } }
=> #<Proc:0x901756c@(irb):23 (lambda)>

> mult.(2).(3)
=> 6
```

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

```
> mult = lambda { |n1, n2| n1 * n2 }.curry
=> #<Proc:0x901dd40 (lambda)>

> dobro = mult.(2)
=> #<Proc:0x901c058 (lambda)>

> triplo = mult.(3)
=> #<Proc:0x9026904 (lambda)>

> dobro.(8)
=> 16

> triplo.(9)
=> 27
```

3.6 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, 2, 3, 4, 5].each { |e| puts "o array contem o numero #{e}" }

=> array contem o numero 1
=> array contem o numero 2
=> array contem o numero 3
=> array contem o numero 4
=> 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, 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..5).each { |e| puts "a range contem o numero #{e}" }

=> range contem o numero 1
=> range contem o numero 2
=> range contem o numero 3
=> range contem o numero 4
=> 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:

```
5.times { |e| puts "numero #{e}" }

=> numero 0
=> numero 1
=> numero 2
=> numero 3
=> 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..21).step(2).each { |e| puts "numero #{e}" }

=> numero 1
=> numero 3
=> numero 5
=> numero 7
=> numero 9
=> numero 11
=> numero 13
=> numero 15
=> numero 17
=> numero 19
=> numero 21
```

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

```
> col = %w(uma lista de Strings para mostrar o for)
> for str in col
>     puts str
> end

=> uma
=> lista
=> de
=> Strings
=> para
=> mostrar
=> o
=> for
```

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

> Podemos utilizar ao invés de `select`, `filter`, que é apenas um *alias* que pode ficar mais confortável para quem está chegando agora vindo de outras linguagens.

Vamos testar com uma `Hash`:

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

Temos também o conceito de *lazy evaluation*. Reparem no método `lazy` antes do `select`:

```
natural_numbers = Enumerator.new do |yielder|
    number = 1

    loop do
        yielder.yield number
        number += 1
    end
end

p natural_numbers.lazy.select { |n| n.odd? }.take(5).to_a
=> [1, 3, 5, 7, 9]
```

> Vamos ver esse tal de `yield` mais para frente. Por enquanto pensem que ele está passando aquele número para frente, para o `Enumerator` criado.

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.

3.6.2 Selecionando os elementos que não atendem uma condição

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

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

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

3.6.3 Processando e alterando os elementos

Vamos alterar os elementos do objeto com o método `map`, primeiro transformando no dobro do valor de cada elemento:

```
> (0..10).map { |valor| valor * 2 }
=> [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Aqui podemos ver que temos alguns conceitos matemáticos envolvidos. Essa operação acima poderia ser descrita como dobrar cada elemento da seleção dos elementos de um conjunto (que é a `Range`) que fazem parte dos números naturais representados aqui pelo símbolo $\mathbb{N}$.

Os números naturais são considerados sendo os números inteiros positivos e utilizamos aqui o número 1 em sobreescrito para deixar claro que o número 0 não está incluído no conjunto, agora o definindo como os números inteiros, selecionando os números menores ou iguais a 10:

$$S = \{2.x | x \in \mathbb{N}_1, x \leq 10\}$$

Podemos deixar mais explícito que queremos os inteiros utilizando $\mathbb{Z}$:

$$S = \{2.x | x \in \mathbb{Z}, x \leq 10\}$$

Esse tipo de operação é conhecido geralmente em matemática como notação de definição de conjunto (*set builder, set comprehension*), e em linguagens de programação como compreensão de lista(*list comprehension*) [a], sendo:

- 2.x = função de saída
- x = variável
- $\mathbb{Z}$ = conjunto de entrada
- x $\leq$ 10 = predicado

[a] https://pt.wikipedia.org/wiki/Compreens%C3%A3o_de_lista

E agora transformando os elementos com um bloco customizado:

```
> %w(um dois tres quatro cinco seis sete oito nove dez).map { |valor| "numero #{valor}" }
=> ["numero um", "numero dois", "numero tres", "numero quatro",
%"numero cinco", "numero seis", "numero sete", "numero oito", "numero nove",
%"numero dez"]

> { 1 => "um", 2 => "dois", 3 => "tres" }.map { |chave, valor| "numero #{valor}" }
=> ["numero um", "numero dois", "numero tres"]
```

3.6.4 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?`:

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

> (0..10).all? { |valor| valor > 0 }
=> false
```

3.6.5 Detectando se algum elemento atende uma condição

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

```
> (0..10).any? { |valor| valor == 3 }
=> true

> (0..10).any? { |valor| valor == 30 }
=> false
```

Nesse caso específico, poderíamos ter escrito dessa forma também:

```
> (0..10).include?(3)
=> true

> (0..10).include?(30)
=> 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.

3.6.6 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?`:

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

3.6.7 Detectando os valores máximo e mínimo

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

```
> (0..10).max
=> 10

> (0..10).min
=> 0
```

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

```
> %w(joao maria antonio).max { |elemento1, elemento2| elemento1.length <=> elemento2.length }
=> "antonio"

> %w(joao maria antonio).min { |elemento1, elemento2| elemento1.length <=> elemento2.length }
=> "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 <=> 2 => -1
1 <=> 1 => 0
1 <=> -1 => 1
```

Olhem que interessante comparando valores de `Hashes`:

```
> { :joao => 33, :maria => 30, :antonio => 25 }.max { |elemento1, elemento2| elemento1[1] <=> elemento2[1] }
=> [:joao, 33]

> { :joao => 33, :maria => 30, :antonio => 25 }.min { |elemento1, elemento2| elemento1[1] <=> elemento2[1] }
=> [:antonio, 25]
```

Desafio 2

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

A partir da versão 2.4, a diferença entre os métodos `min` e `max` é brutal. Vamos rodar o seguinte código em ambas as versões (ok, a parte de *benchmarks* ainda está bem na frente aqui no livro, mas vamos considerar somente os resultados aqui):

```
require 'benchmark'

a = (1..1_000_000).to_a.shuffle

Benchmark.bm do |x|
  x.report("min:") { 1000.times { a.min } }
  x.report("max:") { 1000.times { a.max } }
end
```

Código 43: Comparando valores mínimos e máximos de uma Range

Primeiro no Ruby 2.3.x:

```
$ ruby code/basico/minmax.rb
          user     system      total        real
min: 60.410000   0.020000  60.430000 ( 60.438673)
max: 59.420000   0.030000  59.450000 ( 59.461824)
```

Agora no Ruby 2.4.x:

```
$ ruby code/basico/minmax.rb
          user     system      total        real
min:  1.750000   0.000000   1.750000 (  1.753964)
max:  1.940000   0.000000   1.940000 (  1.943247)
```

E Ruby 3.0.0:

```
       user     system      total        real
min:  0.981920   0.000335   0.982255 (  0.982996)
max:  0.726020   0.000000   0.726020 (  0.726617)
```

Uau. De 60 segundos para menos de 1! Esses métodos estão sendo aprimorados desde a versão 2.4, utilizando métodos novos e mais eficientes de comparação. Alguns dos *commits* podem ser vistos aqui e aqui.

Um método que utiliza o operador `<=>` é o método `clamp`, que tem no seu retorno a garantia que um determinado valor esteja em uma determinada `Range`. Por exemplo:

```
> 10.clamp(1, 5)
 => 5

> 10.clamp(5, 15)
 => 10

> 10.clamp(15, 25)
 => 15
```

Ali aconteceu o seguinte:

1. Foi tentado encaixar 10 entre 1 e 5. Como 10 não está na faixa, foi retornado o (maior) número mais próximo de 10, 5.
2. Foi tentado encaixar 10 entre 5 e 15, o que é perfeitamente possível, retonando 10.

3. Foi tentado encaixar 10 entre 15 e 25. Como 10 não está na faixa, foi retornado o (menor) número mais próximo de 10, 15.

Isso funciona com qualquer tipo de objeto e `Ranges`, como por exemplo com `Strings`:

```
> "c".clamp("a" .. "e")
 => "c"

> "c".clamp("e" .. "f")
 => "e"

> "c".clamp("a" .. "b")
 => "b"
```

3.6.8 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`:

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

Podemos passar também um valor inicial:

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

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

```
> (0..10).inject(:+)
=> 55

> (0..10).inject(100, :+)
=> 155
```

Para o pessoal que adora `JavaScript`, temos um *alias* simpático para `inject`, `reduce`:

```
> (0..10).reduce(:+)
=> 55

> (0..10).reduce(100, :+)
=> 155
```

E a partir da versão `2.4`, temos o método `sum`:

```
> (1..10).sum
=> 55

> (1..10).sum(100)
=> 155
```

Aqui outra curiosidade matemática. Para calcular a soma de um intervalo de inteiros, a linguagem utiliza um método descoberto por Carl Friedrich Gauss quando, segundo reza a lenda do alto dos seus 8 anos de idade, ele respondeu muito rápido na escola qual era a soma dos 100 primeiros números naturais ($\mathbb{N}$). O que ele descobriu na sua cabecinha foi essa fórmula:

Basicamente é: multiple o maior número do intervalo por ele mesmo mais 1 e divida por 2.

$$S = \frac{n(n+1)}{2}$$

Podemos calcular dessa forma e até se o menor número não for 1, utiizando o seguinte código:

```
start_val = 1
end_val   = 10
((end_val + start_val) * (end_val - start_val + 1)) / 2
```

3.6.9 Dividir a coleção em dois Arrays obedecendo uma condição

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

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

3.6.10 Percorrendo os elementos com os índices

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

```
> (0..10).each_with_index { |item, indice| puts "#{item} indice #{indice}" }

=> 0 indice 0
=> 1 indice 1
=> 2 indice 2
=> 3 indice 3
=> 4 indice 4
=> 5 indice 5
=> 6 indice 6
=> 7 indice 7
=> 8 indice 8
=> 9 indice 9
=> 10 indice 10
```

3.6.11 Ordenando uma coleção

Vamos ordenar um `Array` de Strings usando `sort`:

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

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

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

3.6.12 Combinando elementos

Podemos combinar elementos com o método `zip`:

```
> (1..10).zip((11..20))
=> [[1, 11], [2, 12], [3, 13], [4, 14], [5, 15], [6, 16], [7, 17], [8, 18], [9,
19], [10, 20]]

> (1..10).zip((11..20),(21..30))
=> [[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]]
```

Também podemos usar `combination`:

```
> a = %w(john paul george ringo)
=> ["john", "paul", "george", "ringo"]

> a.combination(2)
=> #<Enumerable::Enumerator:0xb7d711a0>

> a.combination(2).to_a
=> [["john", "paul"], ["john", "george"], ["john", "ringo"], ["paul", "george"],
["paul", "ringo"], ["george", "ringo"]]

a.combination(2) { |comb| puts "combinando #{comb[0]} com #{comb[1]}" }

=> combinando john com paul
=> combinando john com george
=> combinando john com ringo
=> combinando paul com george
=> combinando paul com ringo
=> combinando george com ringo
```

Ou `permutation`:

```
> a = %w(john paul george ringo)
=> ["john", "paul", "george", "ringo"]

> a.permutation(2)
=> #<Enumerable::Enumerator:0xb7ce41c4>

> a.permutation(2).to_a
=> [["john", "paul"], ["john", "george"], ["john", "ringo"], ["paul", "john"],
["paul", "george"], ["paul", "ringo"], ["george", "john"], ["george", "paul"],
["george", "ringo"], ["ringo", "john"], ["ringo", "paul"], ["ringo", "george"]]

> a.permutation(2) { |comb| puts "combinando #{comb[0]} com #{comb[1]}" }

=> combinando john com paul
=> combinando john com george
=> combinando john com ringo
=> combinando paul com john
=> combinando paul com george
=> combinando paul com ringo
=> combinando george com john
=> combinando george com paul
=> combinando george com ringo
=> combinando ringo com john
=> combinando ringo com paul
=> combinando ringo com george
```

Ou `product`:

```
> beatles = %w(john paul george ringo)
=> ["john", "paul", "george", "ringo"]

> stooges = %w(moe larry curly shemp)
=> ["moe", "larry", "curly", "shemp"]

beatles.product(stooges)
=> [["john", "moe"], ["john", "larry"], ["john", "curly"], ["john", "shemp"],
["paul", "moe"], ["paul", "larry"], ["paul", "curly"], ["paul", "shemp"],
["george", "moe"], ["george", "larry"], ["george", "curly"], ["george",
"shemp"], ["ringo", "moe"], ["ringo", "larry"], ["ringo", "curly"], ["ringo",
"shemp"]]
```

3.6.13 Percorrendo valores para cima e para baixo

Podemos usar `upto`, `downto` e `step`:

```
> 1.upto(5) { |num| print num, " " }
=> 1 2 3 4 5
=> 1

> 5.downto(1) { |num| print num, " " }
=> 5 4 3 2 1
=> 5

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

3.6.14 Filtrando com o grep

Um método muito útil para coleções é o método `grep` (mesmo nome do utilitário de linha de comando - muito útil, por sinal). Podemos, por exemplo, encontrar determinadas `Strings` em um `Array`, no exemplo abaixo, todas as que tem comprimento entre 3 e 7 caracteres:

```
> %w(eustaquio rangel).grep(/\A\w{3,7}\z/)
=> ["rangel"]
```

Selecionar todos os elementos que sejam iguais ao informado:

```
> [1, 0, 1, 1, 0].grep(0)
=> [0, 0]
```

Encontrar os objetos de uma determinada classe ou módulo:

```
> [1, "String", 1.23, :aqui].grep(Numeric)
=> [1, 1.23]
```

Selecionar os valores de uma determinada faixa, no exemplo abaixo, criando um `Array` com 10 elementos preechidos por números de até 10, selecionando somente os únicos que estão entre 5 e 10:

```
> Array.new(10) { rand(10) }.grep(5..10).uniq
=> [7, 5]
```

Utilizando uma `lambda` para selecionar determinada condição (no exemplo, as `Strings` cujo comprimento é maior que 3):

```
> lamb = ->(str) { str.length > 3 }
=> #<Proc:0x00000003101ff0@(irb):47 (lambda)>

> %w(eustaquio taq rangel).grep(lamb)
=> ["eustaquio", "rangel"]
```

E que tal fazer um sorteador de números da Megasena (se alguém ganhar, lembra de mim!) em apenas uma linha?

```
> (1..60).to_a.shuffle.take(6)
=> [47, 8, 49, 19, 58, 22]

> (1..60).to_a.shuffle.take(6)
=> [38, 17, 16, 29, 28, 37]

> (1..60).to_a.shuffle.take(6)
=> [20, 28, 30, 16, 43, 52]
```

3.6.15 Encadeamento de iteradores

Podemos encadear um iterador direto com outro. Digamos que queremos selecionar os números pares entre 0 e 10 e multiplicar cada um por 2. Podemos utilizar:

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

Podemos encadear quantos iteradores quisermos, mas temos que tomar cuidado para o nosso código não vire uma lista interminável. Fica como dica alinhar um iterador embaixo do outro, a partir do ponto (.), como forma de deixar o código mais legível.

Uma observação importante aqui: Ruby não força o programador a utilizar um determinado estilo para o código, como um determinado número de espaços por tabulação, espaços versus tabulação, identação etc. Tem vantagens e desvantagens, na minha opinião, sendo que a maior desvantagem é que pessoal de desenvolvimento de software é meio complicado e podem ficar horas ou até dias reescrevendo determinado código para o estilo que melhor se convém. O problema que logo após vem outro que tem um estilo totalmente diferente e reescreve novamente e ficam gastando tempo com isso ao invés de agregarem valor ao software fazendo algo mais produtivo do que mudar o estilo de codificação para o seu preferido.

Para evitar isso, podemos utilizar alguns recursos como o Rubocop, que é um *linter* que vai verificar o código escrito de acordo com determinadas regras definidas no Ruby Style Guide e vai dar sugestões de como alterar o código para entrar em conformidade com essas regras. Pegando como exemplo esse código desalinhado que está gravado em um arquivo chamado `select.rb` e rodando o comando `rubocop`, podemos ver que levamos algumas broncas:

```
(0..10).select { |num| num.even? }
    .map { |num| num * 2 }


$ rubocop select.rb
Inspecting 1 file
C

Offenses:

select.rb:1:1: C: [Correctable] Sorbet/FalseSigil: No Sorbet sigil found in
file. Try a typed: false to start (you can also use rubocop -a to automatically
add this).
(0..10).select { |num| num.even? }
```

```
^
select.rb:1:1: C: [Correctable] Style/FrozenStringLiteralComment: Missing frozen
string literal comment.
(0..10).select { |num| num.even? }
^
select.rb:1:16: C: [Correctable] Style/SymbolProc: Pass &:even? as an argument
to select instead of a block.
(0..10).select { |num| num.even? }
               ^^^^^^^^^^^^^^^^^^
select.rb:2:5: C: [Correctable] Layout/MultilineMethodCallIndentation: Align
.map with .select on line 1.
    .map { |num| num * 2 }
    ^^^^
```

O `Rubocop` também é uma formatação de código e ali acima já foram indicadas que temos 4 ofensas de código que podem ser auto-corrigidas. Vamos pedir para serem corrigidas rodando `rubocop -A select.rb` (temos as opções `-a`, menos invasiva, e `-A` que é a completa) e ver o resultado:

```
$ rubocop -a select.rb
Inspecting 1 file
...
1 file inspected, 9 offenses detected, 9 offenses corrected
```

Vejam que agora foram detectadas 9 ofensas, sendo que algumas foram sendo criadas pelo processo mesmo de correção, mas foram todas corrigidas. Dando uma olhada no arquivo agora, já ficou bem diferente. Não esquentem a cabeça se não entenderem uma coisa ou outra ali por enquanto:

```
# typed: false
# frozen_string_literal: true

(0..10).select(&:even?)
       .map { |num| num * 2 }
```

O `Rubocop` é instalado como uma *gem* separada, mais para frente do livro vamos ver como fazer isso. Ele também pode ser embutido direto dentro do seu editor de código preferido para já mostrar as ofensas, as sugestões de correção e até implementar as correções de forma automática quando o arquivo for gravado. Se você utiliza o editor de texto `Vim`, eu recomendo o *plugins* ALE (Asynchronous Lint Engine).

> Se você utiliza o editor de texto `Vim`, dê uma olhada na minha apresentação "Conhecendo o Vim"[a], onde eu listo vários *plugins* interessantes e dicas para o editor.
>
> [a] https://speakerdeck.com/taq/conhecendo-o-vim

> A definição de *linters* vem da ferramenta `lint`, criada em 1978 por por Stephen C. Johnson, um cientista da computação do Bell Labs, enquanto estava lidando em C com problemas de portabilidade de código da plataforma Unix para máquinas de 32 bits.

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`.

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, utilizando o método `tap`:

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

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

A partir da versão `2.7` temos o método `filter_map`, que faz as operações de `select` e `map` em uma passada só:

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

Números randômicos

Podemos gerar números randômicos utilizando a classe `Random`, que é a interface para o `PRNG` (pseudo-random number generator) da linguagem. Vamos ver alguns exemplos:

```
# gerando um número
> Random.rand
 => 0.2844058702317137

# gerando um número inteiro entre 0 10
> Random.rand(10)
 => 6

# gerando um número inteiro entre 5 e 10
> Random.rand(5..10)
 => 7

# gerando um número inteiro entre -10 e 10
> Random.rand(-10..10)
 => -9

# gerando um float até 3.5
> Random.rand(3.5)
 => 2.3398738995523587

# gerando um float entre 15 e 3.5
> Random.rand(1.5 .. 3.5)
 => 3.1253921595006267

 # gerando uma String binária de 5 caracteres
 > Random.bytes(5)
 => "\xB7\xE5\xA7E\xA1"
```

3.7 Métodos

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

```
def diga_oi
   puts "Oi!"
end

diga_oi
=> "Oi!"
```

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

> **Dica**
>
> Podemos definir métodos curtos com a sintaxe de definição de *endless methods*, como nesse caso em que reescrevemos o método acima:
>
> ```
> def diga_oi = puts("oi")
> ```

3.7.1 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:

```
def vezes(p1, p2)
    p1 * p2
end

puts vezes(2, 3)
=> 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.

> **Dica**
>
> Reescrevendo o método acima como *endless method*:
>
> ```
> def vezes(p1, p2) = p1 * p2
> ```

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

```
def cinco_multiplos(numero)
    (1..5).map { |valor| valor * numero }
end

v1, v2, v3, v4, v5 = cinco_multiplos(5)
puts "#{v1}, #{v2}, #{v3}, #{v4}, #{v5}"

=> 5, 10, 15, 20, 25
```

3.7.2 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:

```
def vezes(n1, n2)
   n1 * n2
end

puts vezes(3, 4)
=> 12
```

Podemos contar quantos parâmetros um método recebe usando `arity`:

```
def vezes(n1, n2)
    n1 * n2
end

puts vezes(3, 4)
puts "o metodo recebe #{method(:vezes).arity} parametros"
```

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

```
def oi(nome = "Forasteiro")
   puts "Oi, #{nome}!"
end

oi("TaQ")
=> Oi, TaQ!

oi
=> Oi, Forasteiro!
```

Podemos fazer uso dos argumentos nomeados (*keyword arguments*), indicando que o método vai receber os seus valores identificados:

```
def mostra(a:, b:)
  puts "a é igual #{a}, b é igual #{b}"
end

mostra(a: 1, b: 2)
=> a é igual 1, b é igual 2

mostra(b: 2, a: 1)
=> a é igual 1, b é igual 2
```

Do modo definido acima, ambos os argumentos são obrigatórios:

```
mostra(b: 2)
ArgumentError: missing keyword: a
from (irb):1:in `mostra'
```

Podemos também especificar valores *default* para eles:

```
def mostra(a: 1, b: 2)
  puts "a é igual #{a}, b é igual #{b}"
end

mostra(b: 2)
=> a é igual 1, b é igual 2
```

E também misturar com os argumentos tradicionais:

```
def mostra(a, b: 2)
  puts "a é igual #{a}, b é igual #{b}"
end

mostra(1, b: 2)
=> a é igual 1, b é igual 2
```

Importante notar que a definição do método retorna um símbolo com o nome do método, o que nos permite chamar ele mais tarde direto por essa referência:

```
met = def mostra(a, b: 2)
  puts "a é igual #{a}, b é igual #{b}"
end

send(met, 1, b: 10)

=> a é igual 1, b é igual 10
```

Podemos também utilizar antes do nome do parâmetro o operador `splat`, demonstrado anteriormente:

```
def varios(*valores)
    valores.each { |valor| puts "valor=#{valor}" }
    puts "-"*25
end

=> varios(1)
=> valor=1
=> -------------------------

=> varios(1,2)
=> valor=1
=> valor=2
=> -------------------------

=> varios(1,2,3)
=> valor=1
=> valor=2
=> valor=3
=> -------------------------
```

Na definição do método o operador foi utilizado para concentrar todos os valores recebidos em um `Array`, como pudemos ver acima. Pensem nele como um buraco negro que suga todos os valores!

E também utilizar *pattern matching* para criarmos variáveis com o argumento enviado no método:

```
def varios(valores)
  valores => { a:, b:, c: }
  puts "Recebi a=#{a}, b=#{b}, c=#{c}"
end
(irb):57: warning: One-line pattern matching is experimental, and the behavior
may change in future versions of Ruby!
 => :varios
varios({ a: 1, b: 2, c: 3 })
Recebi a=1, b=2, c=3
```

3.7.3 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`:

```
def executa_bloco(valor)
    yield(valor)
end

executa_bloco(2) { |valor| puts valor * valor }
=> 4

executa_bloco(3) { |valor| puts valor * valor }
=> 9

executa_bloco(4) { |valor| puts valor * valor }
=> 16
```

O método `yield` irá acionar o bloco enviado como argumento, passando no caso acima o argumento de `valor` para o bloco.

Meio que primo do `yield`, temos o `yield_self`, que pega o próprio valor, executa o bloco enviado e retorna o último retorno do bloco:

```
> 1.yield_self { |num| num + 1 }
 => 2
```

Aparentemente, meio bobinho, mas permite com que possamos algumas coisas mais com ele para nos ajudar a reduzir a complexidade e aumentar a legibilidade do código (apesar que tem gente que o usa para justamente o contrário disso). Vamos pegar esse exemplo onde temos uma lambda para duplicar e outra para exponenciar um valor ao quadrado, onde se invertermos a ordem o valor ficará diferente. Queremos que o resultado final seja 16, duplicando e depois exponenciando o número 2. Se invertermos a ordem dos métodos, o resultado será 8. Vamos utilizar o `yield_self` para executar na ordem desejada no segundo caso:

```
> duplicar = -> (val) { val * 2 }
 => #<Proc:0x000056358a8f1400 (irb):29 (lambda)>

> quadrado = -> (val) { val ** 2 }
 => #<Proc:0x000056358a932ef0 (irb):30 (lambda)>

> duplicar(2)
 => 4

> quadrado(duplicar(2))
 => 16

> 2.yield_self(&duplicar).yield_self(&quadrado)
 => 16
```

Ok, não foi o jeito mais bonito de simplificar o código, expressando a sua ordem, mas que tal utilizarmos o `alias then` para ajudar nisso:

```
> 2.then(&duplicar).then(&quadrado)
 => 16
```

Melhor, não? Pessoal de JavaScript fica até mais confortável com o `then` por causa das `Promises` que existem por lá. Sem as `lambas` ou métodos, poderíamos ter escrito assim também:

```
> 2.then { |num| num * 2 }.then { |num| num ** 2 }
 => 16
```

O que nos evitaria código como:

```
> v = 2
 => 2

> v = v * 2
 => 4

> v = v ** 2
 => 16
```

Dica

Podemos utilizar `pow` ou o operador `**` para retornar o número exponenciado:

```
> 2.pow 2
 => 4

> 2.pow 3
 => 8

> 2 ** 2
 => 4

> 2 ** 3
 => 8
```

Isso é meio parecido também com o recurso de **composição de funções**, que nos permite criar novas *lambdas* a partir de outras. Levando em conta os exemplos acima, podemos criar uma nova *lambda* chamada `dup2` (duplicar ao quadrado) com as anteriores:

```
dup  = ->(x) { x * 2  }
sqr  = ->(x) { x ** 2 }
dup2 = dup >> sqr

puts dup2.call(2)

# seria o mesmo que
puts sqr.call(dup.call(2))
```

Código 44: Composição de funções

Vejam que definimos `dup`, depois `sqr` e criamos `dup2` basicamente injetando `dup` dentro de `sqr`. Vejam que essa é a ordem que elas serão chamadas, ou seja, 2 multiplicado por 2 dá 4, que elevado ao quadrado dá 16. E podemos ir compondo as novas funções com quantas quisermos, levando em consideração que devemos explicitamente chamar essas *lambdas* com o método `call` e temos que tomar cuidado com os argumentos esperados pelas funções compostas. Vamos compor com mais funções fazendo algumas coisas malucas somente para efeitos didáticos, como criar outra função para dividir por 2 e outra para calcular 10% do resultado:

```
dup   = ->(x) { x * 2   }
sqr   = ->(x) { x ** 2  }
half  = ->(x) { x / 2   }
tenp  = ->(x) { x * 0.1 }
crazy = dup >> sqr >> half >> tenp

puts crazy.call(2)
```

Código 45: Mais composição de funções

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

```
def executa_bloco(valor)
    yield(valor) if block_given?
end

executa_bloco(2) { |valor| puts valor * valor }
=> 4

executa_bloco(3)
executa_bloco(4) { |valor| puts valor * valor }
=> 16
```

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

```
def executa_bloco(valor, &proc)
    puts proc.call(valor)
end

executa_bloco(2) { |valor| valor * valor }
=> 4
```

3.7.4 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:

```
def altera!(valor)
   valor.upcase!
end

string = "Oi, mundo!"
altera!(string)
puts string

=> "OI, MUNDO!"
```

Lembrando que se estivermos utilizando objetos congelados, isso **não vai funcionar**! Os métodos que tem comportamento **destrutivo**, como explicado daqui a pouco, que alteram o objeto que foi passado como argumento, devem obedecer à convenção de utilizar um ponto de exclamação (`!`) nos seus nomes e devemos prestar atenção tanto em não enviar objetos congelados para eles como nos seus efeitos colaterais (*side effects*) ao enviar objetos que podem ser modificados e depois apresentar algumas "surpresinhas"não esperadas no decorrer da execução do código.

3.7.5 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`:

```
def soma(valor1, valor2)
    valor1 + valor2
rescue
    nil
end

puts soma(1, 2)
=> 3

puts soma(1, :um)
=> nil
```

Também podemos utilizar o `rescue` direto em um modificador de estrutura:

```
value = soma(1, nil) rescue nil
=> nil
```

Vale aqui lembrar que temos uma diferença de performance utilizando `rescue` dessa maneira, onde podemos utilizar o operador ternário:

```
require 'benchmark'

limit = 1_000_000
str   = nil

Benchmark.bm do |x|
  x.report("rescue") do
    limit.times { str.upcase rescue nil }
  end

  x.report("ternário") do
    limit.times { str ? str.upcase : nil }
  end
end
```

Código 46: Comparando rescue com operador ternário

Rodando o programa, podemos ver que a diferença em utilizar o ternário é mais brutal que escutar "Abyssal Gates", do Krisiun:

```
$ ruby code/basico/rescue.rb
              user     system      total        real
rescue    1.850000   0.000000   1.850000 (  1.858982)
ternário  0.060000   0.000000   0.060000 (  0.058927)
```

Então, se performance é um problema (sempre é de se considerar!), considere em evitar utilizar o `rescue` dessa maneira.

3.7.6 Métodos destrutivos e predicados

Também podemos utilizar os caracteres `!` e `?` no final dos nomes dos nossos métodos. Por convenção, métodos com `!` no final são chamados de **métodos destrutivos** e com `?` no final são chamados de **métodos predicados** e são utilizados para testar algo e devem ter retorno *booleano*, retornando `true` ou `false`:

```
def revup!(str)
    str.reverse!.upcase!
end
str = "teste"

puts str.object_id
=> 74439960

revup!(str)
=> ETSET

puts str
=> ETSET

puts str.object_id
=> 74439960
```

```
def ok?(obj)
   !obj.nil?
end

puts ok?(1)
=> true

puts ok?("um")
=> true

puts ok?(:um)
=> true

puts ok?(nil)
=> false
```

Podemos simular parâmetros nomeados usando uma `Hash`:

```
def test(args)
    one = args[:one]
    two = args[:two]
    puts "one: #{one} two: #{two}"
end

test(one: 1, two: 2)
=> one: 1 two: 2

test(two: 2, one: 1)
=> one: 1 two: 2
```

Mas se quisermos de forma explícita, também temos parâmetros nomeados:

```
def foo(str: "foo", num: 123456)
    [str, num]
end

p foo(str: 'buz', num: 9)
=> ['buz', 9]

p foo(str: 'bar')
=> ['bar', 123456]

p foo(num: 123)
=> ['foo', 123]

p foo
=> ['foo', 123456]

p foo(bar: 'buz')
=> ArgumentError
```

Também podemos capturar um método como se fosse uma `Proc`:

```
class Teste
  def teste(qtde)
    qtde.times { puts "teste!" }
  end
end

t = Teste.new
m = t.method(:teste)
p m
m.(3)
p m.to_proc
```

Código 47: Capturando um método

Rodando o programa:

```
$ ruby capture.rb

#<Method: Teste#teste>
teste!
teste!
teste!
#<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:

```
module Enumerable
  def ∑
    self.inject { |memo, val| memo += val }
  end
end

puts [1, 2, 3].∑
puts (0..3).∑
```

Código 48: Nomes de métodos utilizando UTF-8

Rodando o programa:

```
$ ruby encodingmeth.rb
6
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`.

Capítulo 4

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:

```
class Carro
  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
p corsa
puts corsa
```

Código 49: Primeira classe Carro

Rodando o programa:

```
$ ruby carro1.rb
#<Carro:0x894c674 @marca=:chevrolet, @modelo=:corsa, @cor=:preto, @tanque=50>
#<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*[1], 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 de *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`:

```
string = 'Oi, mundo!'
ObjectSpace.define_finalizer(string, ->(id) { puts "Estou terminando o objeto #{id}" })
```

Código 50: Destrutores

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

[1] CamelCase é a denominação em inglês para a prática de escrever as palavras compostas ou frases, onde cada palavra é iniciada com maiúsculas e unidas sem espaços. https://pt.wikipedia.org/wiki/CamelCase

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

Dica

Podemos pegar um objeto pelo seu `object_id`:

```
> s = "oi"
=> "oi"

> i = s.object_id
=> 80832250

> puts ObjectSpace._id2ref(i)
=> 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. Lembram-se que em conversões utilizamos um método chamado `to_s`, que converte o objeto em uma `String`?

Vamos criar um para a nossa classe:

```
class Carro
  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
p corsa
puts corsa
```

Código 51: Segunda classe Carro

Vamos ver o comportamento nas versões 1.8.x:

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

E agora nas versões 1.9.x:

```
$ rvm 1.9.3
$ ruby carro2.rb
```

```
Marca: chevrolet Modelo: corsa Cor: preto Tanque: 50
Marca: chevrolet Modelo: corsa Cor: preto Tanque: 50
```

E agora nas versões acima da 2.x:

```
$ rvm 2.0.0
$ ruby carro2.rb
#<Carro:0x85808b0 @marca=:chevrolet, @modelo=:corsa, @cor=:preto, @tanque=50>
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 `Integer`, 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:

```
class Carro
  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
puts corsa.marca
```

Código 52: Tentando ler variáveis de instância

Rodando o programa:

```
$ ruby carro3.rb
code/carro3.rb:14:in `<main>': undefined method `marca' for Marca:chevrolet
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`:

```ruby
class Carro
  attr_reader :marca, :modelo, :cor, :tanque

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
puts corsa.marca
```

Código 53: Lendo variáveis de instância

Rodando o programa:

```
$ ruby carro4.rb
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:

```ruby
class Carro
  attr_reader :marca, :modelo, :cor, :tanque
  attr_writer :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
corsa.cor = :branco
puts corsa
```

Código 54: Alterando os valores de variáveis de instância

Rodando o programa:

```
$ ruby carro5.rb
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`:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
corsa.cor = :branco
puts corsa
```

Código 55: Lendo e escrevendo em variáveis de instância

Rodando o programa:

```
$ ruby carro6.rb
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`:

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

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

```ruby
require 'ostruct'

carro = OpenStruct.new

carro.marca  = :chevrolet
carro.modelo = :corsa
carro.cor    = :preto
carro.tanque = 50
p carro
=> #<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:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end

  def galoes
    @tanque / 3.785
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
corsa.cor = :branco
puts corsa.galoes
```

Código 56: Criando atributos virtuais

Rodando o programa:

```
$ ruby carro7.rb
13.21003963011889
```

4.1 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:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)

class Carro
  def novo_metodo
    puts "Novo método!"
  end
end
corsa.novo_metodo

class Carro
  remove_method :novo_metodo
end

corsa.novo_metodo
```

Código 57: Abrindo uma classe

Rodando o programa:

```
$ ruby carro8.rb
Novo método!
code/carro8.rb:30:in `<main>': undefined method `novo_metodo' for
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.

Às vezes temos que testar determinados objetos e métodos verificando antes de eles existem. Podemos ver isso no código abaixo, onde os objetos e métodos são verificados usando primeiro um `if` verificando se não existe alguma referência nula, depois, comentado, o método `try` do `ActiveSupport` do `Rails` (que não está disponível em Ruby puro) e por último o navegador de operação segura `&.`, onde é tentado acessar `objeto&.propriedade`, retornando o valor ou nulo se falhar. Isso é conhecido como **navegação segura**:

```ruby
class Volante
  attr_reader :cor

  def initialize(cor)
    @cor = cor
  end
end

class Carro
  attr_reader :volante

  def initialize(volante)
    @volante = volante
  end
end

volante = Volante.new(:preto)
carro   = Carro.new(volante)

puts carro.volante.cor if carro && carro.volante && carro.volante.cor
#puts carro.try(:volante).try(:cor)
puts carro&.volante&.cor
```

Código 58: Navegação segura

4.2 Aliases

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

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

class Carro
  alias to_s_old to_s

  def to_s
    "Esse é um novo jeito de mostrar isso: #{to_s_old}"
  end
end

carro = Carro.new(:chevrolet, :corsa, :preto, 50)
puts carro
puts carro.to_s_old
```

Código 59: Aliases para métodos

Rodando o programa:

```
$ ruby methalias.rb
Esse é um novo jeito de mostrar isso: Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
```

4.3 Inserindo e removendo métodos

Podemos também inserir um método somente em uma determinada instância:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
gol   = Carro.new(:volks, :gol, :azul, 42)

class << corsa
  def novo_metodo
    puts "Novo método!"
  end
end

corsa.novo_metodo
gol.novo_metodo
```

Código 60: Inserindo métodos em uma instância

Rodando o programa:

```
$ ruby insmethinst.rb
Novo método!
code/insmethinst.rb:28:in `<main>': undefined method `novo_metodo' for
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/método `«`? Hora de algumas explicações sobre **metaclasses**!

4.4 Metaclasses

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

```ruby
class Teste
end

p String.ancestors
p Teste.ancestors
```

Código 61: Ancestrais

Rodando o programa:

```
$ ruby ancestors.rb
[String, Comparable, Object, Kernel, BasicObject]
[Teste, Object, Kernel, BasicObject]
```

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

```
class Teste
end

class OutroTeste < Teste
end

p String.superclass
p Teste.superclass
p OutroTeste.superclass
```

Código 62: Superclasses

```
$ ruby superclasses.rb
Object
Object
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`.

```
> BasicObject.instance_methods
=> [:, :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 **metaclasse**, 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[2]. Primeiro, com a classe:

[2]Metaprogramação é escrever código que manipula a linguagem em *runtime*

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
gol   = Carro.new(:volks, :gol, :azul, 42)

Carro.send(:define_method, "multiplica_tanque") do |valor|
  @tanque * valor
end

puts corsa.multiplica_tanque(2)
puts gol.multiplica_tanque(2)
```

Código 63: Metaprogramação

Rodando o programa:

```
$ ruby carro9.rb
100
84
```

Dica

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

Agora, com as instâncias:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end

corsa = Carro.new(:chevrolet, :corsa, :preto, 50)
gol   = Carro.new(:volks, :gol, :azul, 42)

(class << corsa; self; end).send(:define_method, "multiplica_tanque") do |valor|
  @tanque * valor
end

puts corsa.multiplica_tanque(2)
puts gol.multiplica_tanque(2)
```

Código 64: Metaprogramação nas instâncias

Rodando o programa:

```
100
code/carro10.rb:25:in `<main>': undefined method `multiplica_tanque' for
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":

```ruby
class Teste
  def apenas_uma_vez
    def self.apenas_uma_vez
      raise StandardError, "Esse metodo se destruiu!"
    end

    puts "Vou rodar apenas essa vez hein?"
  end
end

teste = Teste.new
teste.apenas_uma_vez
teste.apenas_uma_vez
```

Código 65: Método auto-destrutivo

Rodando o programa:

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

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

4.5 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:

```
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor
  @@qtde = 0

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    @@qtde += 1
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end

  def qtde
    @@qtde
  end
end

corsa    = Carro.new(:chevrolet, :corsa, :preto, 50)
gol      = Carro.new(:volks, :gol, :azul, 42)
ferrari  = Carro.new(:ferrari, :viper, :vermelho, 70)

puts ferrari.qtde
```

Código 66: Variáveis de classe

Rodando o programa:

```
$ ruby classvar1.rb
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:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor
  @@qtde = 0

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    @@qtde += 1
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end

  def self.qtde
    @@qtde
  end
end

corsa   = Carro.new(:chevrolet, :corsa, :preto, 50)
gol     = Carro.new(:volks, :gol, :azul, 42)
ferrari = Carro.new(:ferrari, :enzo, :vermelho, 70)
puts Carro.qtde
```

Código 67: Criando métodos de classe

Rodando o programa:

```
$ ruby classvar2.rb
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`:

```
public class CarroEstatico {
  private static int qtde = 0;

  public CarroEstatico() {
    ++qtde;
  }

  public static int qtde() {
    return qtde;
  }

  public static void main(String args[]) {
    CarroEstatico[] carros = new CarroEstatico[10];

    for (int i = 0; i < carros.length; i++) {
      carros[i] = new CarroEstatico();
      System.out.println(CarroEstatico.qtde() + " carros");
    }
  }
}
```

Código 68: Primeira classe Carro

Rodando o programa:

```
$ java CarroEstatico
1 carros
2 carros
3 carros
4 carros
5 carros
6 carros
7 carros
8 carros
9 carros
10 carros
```

4.5.1 Interfaces fluentes

O método `self` é particularmente interessante para desenvolvermos *interfaces fluentes*[3], 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:

[3] http://en.wikipedia.org/wiki/Fluent_interface

```
class SQL
  attr_reader :table, :conditions, :order

  def from(table)
    @table = table
    self
  end

  def where(cond)
    @conditions = cond
    self
  end

  def order(order)
    @order = order
    self
  end

  def to_s
    "select * from #{@table} where #{@conditions} order by #{@order}"
  end
end

sql = SQL.new.from("carros").where("marca='Ford'").order("modelo")
puts sql
```

Código 69: Interfaces fluentes

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 encadeiamento foi chamado.

4.6 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:

```ruby
@@qtde = 10

class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor
  @@qtde = 0
  puts self

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    @@qtde += 1
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end

  def self.qtde
    @@qtde
  end
end

puts self
puts @@qtde
```

Código 70: Variáveis de instância de classe

Rodando o programa:

```
$ ruby classvar3.rb
Carro
main
0
```

Alerta

A partir da versão 2.0, rodar esse programa nos retorna um *warning*:

```
$ ruby classvar3.rb
classvar3.rb:2: warning: class variable access from toplevel
Carro
main
classvar3.rb:28: warning: class variable access from toplevel
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** :

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  class << self
    attr_accessor :qtde
  end
  @qtde = 0

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    self.class.qtde += 1
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end

corsa   = Carro.new(:chevrolet, :corsa, :preto, 50)
gol     = Carro.new(:volks ,:gol, :azul, 42)
ferrari = Carro.new(:ferrari, :enzo, :vermelho, 70)
puts Carro.qtde
```

Código 71: Variáveis de instância de classe

```
$ ruby classvar4.rb
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 `@@`).

4.7 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++:

```cpp
#include <iostream>

using namespace std;

class Automovel {
  public:
    void ligar() {
      cout << "ligando o automóvel\n";
    }
};

class Radio {
  public:
    void ligar() {
      cout << "ligando o rádio\n";
    }
};

class Carro: public Automovel, public Radio {
  public:
    Carro() {}
};

int main() {
  Carro carro;
  carro.ligar(); // só compila com Automovel::ligar();
  return 0;
}
```

Código 72: Carro em C++

Se compilarmos esse código, vamos ter esse resultado:

```
$ g++ -g -o carro carro.cpp
carro.cpp: Na função ‘int main()’:
carro.cpp:26:10: erro: request for member ‘ligar’ is ambiguous
carro.cpp:14:9: erro: candidates are: void Radio::ligar()
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

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

para

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

que resulta em

```
$ g++ -g -o carro carro.cpp
$ ./carro
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>`:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor
  @@qtde = 0

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    @@qtde += 1
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end

  def self.qtde
    @@qtde
  end
end

class NovoCarro < Carro
  def to_s
    "Marca nova:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end

carro1 = Carro.new(:chevrolet, :corsa, :preto, 50)
carro2 = Carro.new(:chevrolet, :corsa, :prata, 50)
novo_carro = NovoCarro.new(:volks, :gol, :azul, 42)

puts carro1
puts carro2
puts novo_carro
puts Carro.qtde
puts NovoCarro.qtde
```

Código 73: Herança

Rodando o programa:

```
$ ruby carro11.rb
Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
Marca:chevrolet Modelo:corsa Cor:prata Tanque:50
Marca nova:volks Modelo:gol Cor:azul Tanque:42
3
3
```

Poderíamos ter modificado para usar o método `super`:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor
  @@qtde = 0

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    @@qtde += 1
  end

  def to_s
    "Marca: #{@marca} Modelo: #{@modelo} Cor: #{@cor} Tanque: #{@tanque}"
  end
end

class NovoCarro < Carro
  def to_s
    "Novo Carro: #{super}"
  end
end

carro      = Carro.new(:chevrolet, :corsa, :preto, 50)
novo_carro = NovoCarro.new(:volks, :gol, :azul, 42)

puts carro
puts novo_carro
```

Código 74: Método super

Rodando o programa:

```
$ ruby carro12.rb
Marca:chevrolet Modelo:corsa Cor:preto Tanque:50
Novo Carro: Marca:volks Modelo:gol Cor:azul Tanque:42
```

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

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

Podemos ver como enviar só os selecionados:

```
class Teste
  def metodo(parametro1)
    puts parametro1
  end
end

class NovoTeste < Teste
  def metodo(parametro1, parametro2)
    super(parametro1)
    puts parametro2
  end
end

t1 = Teste.new
t2 = NovoTeste.new
t1.metodo(1)
t2.metodo(2,3)
```

Código 75: Método super com argumentos selecionados

Rodando o programa:

```
$ ruby supermeth.rb
1
2
3
```

Dica

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

```
class Pai
   def self.inherited(child)
       puts "#{child} herdando de #{self}"
   end
end

class Filha < Pai
end


$ ruby inherited.rb
Filha herdando de Pai
```

Dica

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

```
class Carro < Struct.new(:marca, :modelo, :cor, :tanque)
   def to_s
       "Marca: #{marca} modelo: #{modelo} cor: #{cor} tanque: #{tanque}"
   end
end
fox = Carro.new(:vw, :fox, :verde, 45)
puts fox
=> Marca: vw modelo: fox cor: verde tanque: 45
```

4.8 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:

```
> c1 = Carro.new
=> #<Carro:0x9f0e138>

> c2 = c1
=> #<Carro:0x9f0e138>

> c3 = c1.dup
=> #<Carro:0x9f1d41c>

> c1.object_id
=> 83390620

> c2.object_id
=> 83390620

> c3.object_id
=> 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:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque, :criado
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    @criado = Time.now
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end

carro = Carro.new(:chevrolet, :corsa, :preto, 50)
puts carro.criado
sleep 1

outro_carro = carro.dup
puts outro_carro.criado
```

Código 76: Duplicando de modo raso

Rodando o programa:

```
$ruby dup.rb
2016-06-29 22:36:10 -0300
2016-06-29 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`:

```ruby
class Carro
  attr_reader :marca, :modelo, :tanque, :criado
  attr_accessor :cor

  def initialize(marca, modelo, cor, tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
    @criado = Time.now
  end

  def initialize_copy(original)
    puts "criado objeto novo #{self.object_id} duplicado de #{original.object_id}"
    @criado = Time.now
  end

  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end

carro = Carro.new(:chevrolet, :corsa, :preto, 50)
puts carro.criado
puts carro.object_id
sleep 1

outro_carro = carro.dup
puts outro_carro.criado
puts outro_carro.object_id
```

Código 77: Customizando o objeto duplicado

Rodando o programa:

```
$ ruby initializecopy.rb
2016-06-29 22:36:10 -0300
83042330
criado objeto novo 82411250 duplicado de 83042330
2016-06-29 22:36:11 -0300
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:

```ruby
class A
  attr_reader :outro

  def initialize(outro = nil)
    @outro = outro
  end

  def show
    puts "Estou em #{self.class.name}, #{object_id}"
    puts "Outro: #{@outro.object_id}" if !@outro.nil?
  end
end

class B < A
end

a = A.new
b = B.new(a)

a.show
b.show

b2 = b.dup
b2.show
```

Código 78: Shallow copy

Rodando o programa:

```
$ ruby shallow.rb
Estou em A, 75626430
Estou em B, 75626420
Outro: 75626430 <===== aqui!
Estou em B, 75626300
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:

```ruby
class A
  attr_accessor :outro

  def initialize(outro = nil)
    @outro = outro
  end

  def show
    puts "Estou em #{self.class.name}, #{object_id}"
    puts "Outro: #{@outro.object_id}" if !@outro.nil?
  end
end

class B < A
end

a = A.new
b = B.new(a)

a.show
b.show

b2 = Marshal.load(Marshal.dump(b))
b2.show
```

Código 79: Deep copy

Rodando o programa:

```
$ ruby deep.rb
Estou em A, 74010500
Estou em B, 74010490
Outro: 74010500 <===== aqui!
Estou em B, 74010330
Outro: 74010300 <===== aqui!
```

4.8.1 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:

```ruby
class Teste
  def method_missing(meth, value = nil)
    sanitized = meth.to_s.split("=").first

    if meth =~ /=$/
      self.class.send(:define_method, meth) { |val| instance_variable_set("@#{sanitized}", val) }
      self.send(meth, value)
    else
      self.class.send(:define_method, sanitized) { instance_variable_get("@#{sanitized}") }
      self.send(meth)
    end
  end
end

t = Teste.new
t.oi = "oi, mundo!"
puts t.oi

puts t.hello
t.hello = "hello, world!"
puts t.hello
```

Código 80: Interceptando métodos que não existem

Rodando o programa:

```
$ ruby methmissing.rb
oi, mundo!
hello, world!
```

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

```ruby
class Teste
  def self.method_added(meth)
    puts "Adicionado o método #{meth}"
  end

  def self.method_removed(meth)
    puts "Removido o método #{meth}"
  end
end

t = Teste.new
t.class.send(:define_method,"teste") { puts "teste!" }
t.teste
t.class.send(:remove_method,:teste)
t.teste
```

Código 81: Interceptando métodos adicionados e removidos

Rodando o programa:

```
$ ruby hooksmeth.rb
Adicionado o método teste
teste!
Removido o método teste
code/hooksmeth.rb:16:in `<main>': undefined method `teste' for #<Teste:0x9f3d12c> (NoMethodError)
```

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

```ruby
class Teste
  def method_missing(meth)
    puts "Não sei o que fazer com a sua requisição: #{meth}"
  end
end

t = Teste.new
t.teste
```

Código 82: Métodos fantasmas

```
$ ruby ghost.rb
Não sei o que fazer com a sua requisição: teste
```

4.9 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:

```ruby
class CaixaDeParafusos
  attr_reader :quantidade

  def initialize(quantidade)
    @quantidade = quantidade
  end

  def to_s
    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
  end

  def +(outra)
    CaixaDeParafusos.new(@quantidade + outra.quantidade)
  end
end

caixa1 = CaixaDeParafusos.new(10)
caixa2 = CaixaDeParafusos.new(20)
caixa3 = caixa1 + caixa2

puts caixa1
puts caixa2
puts caixa3
```

Código 83: Somando um objeto com outro

Rodando o programa:

```
$ ruby caixa1.rb
Quantidade de parafusos na caixa 69826490: 10
Quantidade de parafusos na caixa 69826480: 20
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:

```ruby
class CaixaDeParafusos
  attr_reader :quantidade

  def initialize(quantidade)
    @quantidade = quantidade
  end

  def to_s
    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
  end

  def +(outra)
    CaixaDeParafusos.new(@quantidade + outra.quantidade)
    @quantidade = 0
    outra.quantidade = 0
  end
end

caixa1 = CaixaDeParafusos.new(10)
caixa2 = CaixaDeParafusos.new(20)
caixa3 = caixa1 + caixa2

puts caixa1
puts caixa2
puts caixa3
```

Código 84: Somando um objeto com outro e interagindo com eles

Rodando o programa:

```
$ ruby caixa2.rb
code/caixa2.rb:15:in `+': undefined method `quantidade=' for Quantidade de parafusos na caixa 74772290: 20:Ca
    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**:

```
class CaixaDeParafusos
  protected
  attr_writer :quantidade

  public
  attr_reader :quantidade

  def initialize(quantidade)
    @quantidade = quantidade
  end

  def to_s
    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
  end

  def +(outra)
    nova = CaixaDeParafusos.new(@quantidade + outra.quantidade)
    @quantidade = 0
    outra.quantidade = 0
    nova
  end
end

caixa1 = CaixaDeParafusos.new(10)
caixa2 = CaixaDeParafusos.new(20)
caixa3 = caixa1 + caixa2

puts caixa1
puts caixa2
puts caixa3
```

Código 85: Utilizando métodos protegidos

Rodando o programa:

```
$ ruby caixa3.rb

Quantidade de parafusos na caixa 81467020: 0
Quantidade de parafusos na caixa 81467010: 0
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 enjoos 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 /:

```
class CaixaDeParafusos
  protected
  attr_writer :quantidade

  public
  attr_reader :quantidade

  def initialize(quantidade)
    @quantidade = quantidade
  end

  def to_s
    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
  end

  def +(outra)
    nova = CaixaDeParafusos.new(@quantidade + outra.quantidade)
    @quantidade = 0
    outra.quantidade = 0
    nova
  end

  def /(quantidade)
    caixas = Array.new(@quantidade / quantidade, quantidade)
    caixas << @quantidade % quantidade if @quantidade % quantidade > 0
    @quantidade = 0
    caixas.map { |quantidade| CaixaDeParafusos.new(quantidade) }
  end
end

caixa1 = CaixaDeParafusos.new(10)
caixa2 = CaixaDeParafusos.new(20)
caixa3 = caixa1 + caixa2

puts caixa3 / 8
```

Código 86: Dividindo o objeto

```
$ ruby caixa4.rb
Quantidade de parafusos na caixa 67441310: 8
Quantidade de parafusos na caixa 67441300: 8
Quantidade de parafusos na caixa 67441290: 8
Quantidade de parafusos na caixa 67441280: 6
```

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

teúdo:

```
class CaixaDeParafusos
  protected
  attr_writer :quantidade

  public
  attr_reader :quantidade

  def initialize(quantidade)
    @quantidade = quantidade
  end

  def to_s
    "Quantidade de parafusos na caixa #{self.object_id}: #{@quantidade}"
  end

  def +(outra)
    nova = CaixaDeParafusos.new(@quantidade + outra.quantidade)
    @quantidade = 0
    outra.quantidade = 0
    nova
  end

  def /(quantidade)
    caixas = Array.new(quantidade, @quantidade / quantidade)
    (@quantidade % quantidade).times { |indice| caixas[indice] += 1 }
    @quantidade = 0
    caixas.map { |quantidade| CaixaDeParafusos.new(quantidade) }
  end
end

caixa1 = CaixaDeParafusos.new(10)
caixa2 = CaixaDeParafusos.new(20)
caixa3 = caixa1 + caixa2

puts caixa3 / 4
```

Código 87: Divindo o objeto em objetos menores

Rodando o programa:

```
$ ruby caixa5.rb
Quantidade de parafusos na caixa 81385900: 8
Quantidade de parafusos na caixa 81385890: 8
Quantidade de parafusos na caixa 81385880: 7
Quantidade de parafusos na caixa 81385870: 7
```

4.10 Executando blocos em instâncias de objetos

Quando temos uma instância de algum objeto, podemos executar blocos dessa maneira:

```
> i = 1
> i.instance_eval { puts "meu valor é: #{self}" }
=> meu valor é: 1
```

O método `instance_eval` é bem legal, mas ele não recebe argumentos. Por exemplo:

```
> i.instance_eval 10, &->(val){ puts "meu valor é: #{self}, mais #{val} dá #{self + val}" }
=> ArgumentError: wrong number of arguments (1 for 0)
```

> **Dica**
>
> Repararam como eu converti uma `lambda` para um bloco ali acima utilizando `&`?

Para aceitar argumentos, vamos utilizar `instance_exec`:

```
> i.instance_exec 10, &->(val){ puts "meu valor é: #{self}, mais #{val} dá #{self + val}" }
=> meu valor é: 1, mais 10 dá 11
```

4.11 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:

```
def cria_contador(inicial, incremento)
  contador = inicial
  lambda { contador += incremento }
end

meu_contador = cria_contador(0, 1)

puts meu_contador.call
puts meu_contador.call
puts meu_contador.call
```

Código 88: Closures

```
$ ruby closures.rb
1
2
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`.

Capítulo 5

Módulos

5.1 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`:

```
class Primata
  def come
    puts "Nham!"
  end

  def dorme
    puts "Zzzzzz..."
  end
end

class Humano < Primata
  def conecta_na_web
    puts "Login ... senha ..."
  end
end

module Ave
  def voa
    puts "Para o alto, e avante!"
  end
end

class Mutante < Humano
  include Ave
end

mutante = Mutante.new
mutante.come
mutante.dorme
mutante.conecta_na_web
mutante.voa
```

Código 89: Incluindo um módulo

Rodando o programa:

```
$ ruby mod1.rb
Nham!
Zzzzzz...
```

```
Login ... senha ...
Para o alto, e avante!
```

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

```
class Primata
  def come
    puts "Nham!"
  end

  def dorme
    puts "Zzzzzz..."
  end
end

class Humano < Primata
  def conecta_na_web
    puts "Login ... senha ..."
  end
end

module Ave
  def voa
    puts "Para o alto, e avante!"
  end
end

class Mutante < Humano
end

mutante = Mutante.new
mutante.extend(Ave)
mutante.come
mutante.dorme
mutante.conecta_na_web
mutante.voa

mutante2 = Mutante.new
mutante2.voa
```

Código 90: Incluindo um módulo em uma instância

```
$ ruby mod2.rb
Nham!
Zzzzzz...
Login ... senha ...
Para o alto, e avante!
code/mod2.rb:33:in `<main>': undefined method `voa' for #<Mutante:0x855465c> (NoMethodError)
```

Dica

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

Dica

Também podemos incluir o módulo na classe dessa forma:

```
Humano.send(:include, Mutante)
```

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 (???):

```
module TesteMod
  module ClassMethods
    def class_method
      puts "Esse é um método da classe!"
    end
  end

  def self.included(where)
    where.extend(ClassMethods)
  end

  def instance_method
    puts "Esse é um método de instância!"
  end
end

class TesteCls
  include TesteMod
end

t = TesteCls.new
t.instance_method
TesteCls.class_method
```

Código 91: Interceptando a inclusão de um módulo

Rodando o programa:

```
$ ruby mod7.rb
Esse é um método de instância!
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:

```
class CaixaDeParafusos
  include Comparable
  attr_reader :quantidade

  def initialize(quantidade)
    @quantidade = quantidade
  end

  def <=>(outra)
    self.quantidade <=> outra.quantidade
  end
end

caixa1 = CaixaDeParafusos.new(10)
caixa2 = CaixaDeParafusos.new(20)
caixa3 = CaixaDeParafusos.new(10)

puts caixa1 < caixa2
puts caixa2 > caixa3
puts caixa1 == caixa3
puts caixa3 > caixa2
puts caixa1.between?(caixa3, caixa2)
```

Código 92: Comparando usando módulos

Rodando o programa:

```
$ ruby mod3.rb
true
true
true
false
true
```

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

```ruby
class Parafuso
  attr_reader :polegadas

  def initialize(polegadas)
    @polegadas = polegadas
  end

  def <=>(outro)
    self.polegadas <=> outro.polegadas
  end

  def to_s
    "Parafuso #{object_id} com #{@polegadas}\""
  end
end

class CaixaDeParafusos
  include Enumerable

  def initialize
    @parafusos = []
  end

  def <<(parafuso)
    @parafusos << parafuso
  end

  def each
    @parafusos.each { |parafuso| yield(parafuso) }
  end
end

caixa = CaixaDeParafusos.new
caixa << Parafuso.new(1)
caixa << Parafuso.new(2)
caixa << Parafuso.new(3)

puts "o menor parafuso na caixa é: #{caixa.min}"
puts "o maior parafuso na caixa é: #{caixa.max}"
puts "os parafusos com medidas par são: #{caixa.select { |parafuso| parafuso.polegadas % 2 == 0}.join(',')}"
puts "duplicando a caixa: #{caixa.map { |parafuso| Parafuso.new(parafuso.polegadas * 2)}}"
```

Código 93: Iteradores usando um módulo

Rodando o programa:

```
$ ruby mod4.rb
o menor parafuso na caixa é: Parafuso 72203410 com 1"
o maior parafuso na caixa é: Parafuso 72203390 com 3"
os parafusos com medidas par são: Parafuso 72203400 com 2"
duplicando a caixa: [Parafuso 72203110 com 2", Parafuso 72203100 com 4", Parafuso 72203090 com 6"]
```

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

```
class C
  def x
    'x'
  end
end

module M
  def x
    "[#{super}]"
  end

  def y
    'y'
  end
end

class C
  include M
end

p C.ancestors
c = C.new
puts c.x
puts c.y
```

Código 94: Ancestrais com módulos

Rodando o programa:

```
[C, M, Object, Kernel, BasicObject]
x
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.

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

```ruby
class C
  def x
    'x'
  end
end

module M
  def x
    "[#{super}]"
  end

  def y
    'y'
  end
end

class C
  prepend M
end

p C.ancestors # => [M, C, Object, Kernel, BasicObject]
c = C.new

puts c.x # => [x]
puts c.y # => y
```

Código 95: Incluindo um módulo antes na cadeia de métodos

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:

```
module Automovel
  def ligar
    puts "ligando automóvel"
  end
end

module Radio
  def ligar
    puts "ligando rádio"
  end
end

class Carro
  include Automovel
  include Radio
end

c = Carro.new
c.ligar
```

Código 96: Incluindo vários módulos

Rodando o código:

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

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

```
$ Carro.ancestors
=> [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:

```ruby
module Automovel
  def ligar
    puts "Ligando automóvel #{@marca}"
  end
end

module Radio
  def ligar
    puts "Ligando rádio #{@marca}"
  end
end

class Carro
  include Automovel
  include Radio

  def initialize
    @marca = :vw
  end

  def ligar
    Automovel.instance_method(:ligar).bind(self).call
    super
  end
end

Carro.new.ligar
```

Código 97: Utilizando bind

Rodando o programa:

```
Ligando automóvel vw
```

Aqui temos um conceito interessante: o `bind` faz com que "localizemos"o método dentro do contexto de execução corrente, que é a instância do objeto. Dessa forma, podemos acessar a variável de instância `marca`, assim como o método que foi inserido por último. Se não utilizamos `bind`, vamos receber um erro.

Temos também o método `binding`, que retorna o contexto de execução corrente, o que permite que alguns códigos possam ser executados dentro do mesmo, como esse exemplo do `ERB`:

```ruby
require 'erb'

class User
  def initialize
    @name = 'taq'
  end

  def resolve_binding
    binding
  end
end

user = User.new
erb  = ERB.new "Olá, <%= @name %>!"

puts erb.result(user.resolve_binding)
```

Código 98: Utilizando binding

Módulos estendendo a si mesmos!

Aqui tem um lance meio *inception*: um módulo pode estender a si mesmo! Imaginem que precisamos de um módulo, *que não precisa de uma instância de um objeto*, que tem alguns métodos que podem ser chamamos como métodos estáticos. Poderíamos ter alguns definidos tradicionalmente como:

```
module Inception
  def self.hello
    puts 'hello'
  end

  def self.world
    puts 'world'
  end
end

Inception.hello
Inception.world
```

Código 99: Módulos estendendo a si mesmos

Mas também podemos escrever isso dessa forma:

```
module Inception
  extend self

  def hello
    puts 'hello'
  end

  def world
    puts 'world'
  end
end

Inception.hello
Inception.world
```

Código 100: Módulos estendendo a si mesmos

No primeiro exemplo, ficou bem claro que os métodos são estáticos, através do uso de `self`, enquanto no segundo, ficou meio "feitiçaria", fazendo com que o `extend self` no início fizesse com que o módulo estendesse a si mesmo, injetando os seus métodos de *instância* (hein?), da sua *eigenclass*, como métodos de *classe* (hein, de novo?)!

O resultado vai ser similar, mas convém analisar a clareza do código levando em conta a visibilidade do primeiro exemplo contrastando com a forma prática, porém "vodu", do segundo.

Podemos implementar algumas funcionalidades interessantes com módulos, por exemplo, criar uma classe `singleton` [1], fazendo uso de um recurso que vamos aprender no próximo capítulo, que é instalar funcionalidades externas para uso na linguagem através das **RubyGems**:

[1] http://pt.wikipedia.org/wiki/Singleton

```
require 'singleton'

class Teste
  include Singleton
end

begin
  Teste.new
rescue StandardError => e
  puts "Não consegui criar usando new: #{e}"
end

puts Teste.instance.object_id
puts Teste.instance.object_id
```

Código 101: Implementando singletons

Rodando o programa:

```
$ ruby mod6.rb
Não consegui criar usando new: private method ‘new’ called for Teste:Class
69705530
69705530
```

Uma classe `singleton` basicamente só permite que seja criada uma instância só de uma classe, deixando o método construtor acessível somente de forma privada para a classe, armazenando o objeto novo internamente pelo construtor e o retornando com o método `instance`. Esse comportamento é bem interessante para a remoção de objetos de escopo global, pois fica contido e menos exposto.

5.2 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**:

```
module Paulista
  class Pessoa
    def comida_preferida
      "pizza"
    end
  end
end

module Gaucho
  class Pessoa
    def comida_preferida
      "churrasco"
    end
  end
end

pessoa1 = Paulista::Pessoa.new
pessoa2 = Gaucho::Pessoa.new

puts pessoa1.comida_preferida
puts pessoa2.comida_preferida
```

Código 102: Namespaces

Rodando o programa:

```
$ ruby mod5.rb
pizza
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[2], utilizando a *keyword* `package`, como por exemplo:

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

Dando uma olhada em como resolvemos isso em Java:

```
// localizado em com/eustaquiorangel/paulista/Pessoa.java
package com.eustaquiorangel.paulista;

public class Pessoa {
  public static String comidaPreferida() {
    return "pizza";
  }
}
```

Código 103: Pessoa, paulista, em Java

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:

[2] http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

```java
// localizado em com/eustaquiorangel/gaucho/Pessoa.java
package com.eustaquiorangel.gaucho;

public class Pessoa {
  public static String comidaPreferida() {
    return "churrasco";
  }
}
```

Código 104: Pessoa, gaúcha, em Java

```java
/**
 * Exemplo de namespace utilizando duas classes com o mesmo nome, mas com
 * namespaces diferentes
*/

public class Namespace {
  public static void main(String args[]) {
    System.out.println(com.eustaquiorangel.paulista.Pessoa.comidaPreferida());
    System.out.println(com.eustaquiorangel.gaucho.Pessoa.comidaPreferida());
  }
}
```

Código 105: Namespaces em Java

```
$ javac -cp .:./* Namespace.java
$ java -cp .:./* Namespace
pizza
churrasco
```

5.3 TracePoint

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`:

```
TracePoint.new(:class,:end,:call) do |tp|
  puts "Disparado por #{tp.self} no arquivo #{tp.path} na linha #{tp.lineno}"
end.enable

module Paulista
  class Pessoa
  end
end
p = Paulista::Pessoa.new
```

Código 106: TracePoint

Rodando o programa:

```
$ ruby tpoint.rb
Disparado por Paulista no arquivo tpoint.rb na linha 5
Disparado por Paulista::Pessoa no arquivo tpoint.rb na linha 6
Disparado por Paulista::Pessoa no arquivo tpoint.rb na linha 7
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 um exemplo para tentar definir uma interface em Ruby, e dei uma mexida nele para ficar assim:

```
module AbstractInterface
  class NotImplementedError < StandardError
    def initialize(*methods)
      super "You must implement the following methods: #{methods.join(', ')}"
    end
  end

  def AbstractInterface.check_methods(klass, other, methods)
    return if other.class == Module

    TracePoint.new(:end) do |tp|
      return if tp.self != other || methods.nil?
      missing = methods.select { |method| !other.instance_methods.include?(method) }
      raise NotImplementedError.new(missing) if missing.any?
    end.enable
  end

  module ClassMethods
    def abstract_method(*args)
      return @abstract_method if !args
      @abstract_method ||= []
      @abstract_method.push(*args)
    end

    def included(other)
      AbstractInterface.check_methods(self, other, @abstract_method)
    end

    def check_methods(klass, other, methods)
      AbstractInterface.check_methods(klass, other, methods)
    end
  end

  def self.included(other)
    check_methods(self, other, @abstract_method)
    other.extend ClassMethods
  end
end
```

Código 107: Interfaces

Agora utilizando:

```
module FooBarInterface
  include AbstractInterface
  abstract_method :foo, :bar
end

module BazInterface
  include AbstractInterface
  abstract_method :baz
end

class Test
  include FooBarInterface
  include BazInterface

  def foo
    puts "foo"
```

```
  end

  def bar
    puts "bar"
  end

  def baz
    puts "baz"
  end
end

t = Test.new
t.foo
t.bar
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 os pacotes novos que vão nos prover essas funcionalidades, através das **RubyGems**.

Capítulo 6

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 https://rubygems.org/pages/download?locale=pt-BR e executar os comandos seguintes como *root* ou usando *sudo*), levando em conta que `<versao>` é o número da versão disponível na URL acima:

```
$ wget https://rubygems.org/rubygems/rubygems-<versao>.tgz
$ tar xvzf rubygems-<versao>.tgz
$ cd rubygems-<versao>
$ ruby setup.rb
$ gem -v => <versao>
```

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

Mas novamente: isso quase nunca é necessário. Vamos ter disponível o sistema de *gems* pela RVM ou pelos pacotes do sistema operacional.

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:

  ```
  $ gem install memoize
  Successfully installed memoize-<versao>
  Installing ri documentation for memoize-<versao>...
  Installing RDoc documentation for memoize-<versao>...
  ```

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

```
$ gem search -l memo
*** LOCAL GEMS ***
memoize (<versao>)
```

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:

```
$ gem update --system
```

Instalamos essa *gem* especifica 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

```
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. Mas já fica a dica se o seu projeto está com uma versão menor que a 1.9.x, tem alguma coisa errada por aí.

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`, no arquivo `load.1.rb`:

```
class Teste
   def initialize
       puts "comportamento padrão"
   end
end

load("load2.rb")
Teste.new
```

E agora `load2.rb`:

```
class Teste
   def initialize
       puts "comportamento reescrito"
   end
end
```

Rodando:

```
$ ruby load1.rb
=> comportamento reescrito
```

Mas se agora trocarmos e utilizarmos

```
load("load2.rb", true)
```

:

```
$ ruby load1.rb
=> 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[1]? Primeiro vamos ver sem usar *memoization*:

```
require 'benchmark'

def fib(numero)
  return numero if numero < 2
  fib(numero - 1) + fib(numero - 2)
end

puts Benchmark.measure { puts fib(ARGV[0].to_i) }
```

Código 108: Fibonacci sem memoization

Rodando o programa com alguns números de entrada:

```
$ ruby memo1.rb 10
55
  0.000023   0.000011   0.000034 (  0.000030)

$ ruby memo1.rb 20
6765
  0.000570   0.000095   0.000665 (  0.000663)

$ ruby memo1.rb 30
832040
  0.080281   0.000000   0.080281 (  0.080310)

$ ruby memo1.rb 40
102334155
  9.718632   0.000000   9.718632 (  9.723784)
```

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

```
public class Fib {
  public static long calcula(int numero) {
    if (numero < 2) {
      return numero;
    }
    return calcula(numero - 1) + calcula(numero - 2);
  }

  public static void main(String args[]) {
      long started_at = System.currentTimeMillis();
      System.out.println(calcula(Integer.parseInt(args[0])));

      long diff = System.currentTimeMillis() - started_at;
      System.out.println((float) diff / 1000);
  }
}
```

Código 109: Fibonacci em Java

Rodando o programa:

```
$ java Fib 10
55
0.0

$ java Fib 20
6765
0.0

$ java Fib 30
832040
0.004

$ java Fib 40
102334155
0.409
```

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

```
require 'benchmark'
require 'memoize'
include Memoize

def fib(numero)
  return numero if numero < 2
  fib(numero - 1) + fib(numero - 2)
end
memoize(:fib)

puts Benchmark.measure { puts fib(ARGV[0].to_i) }
```

Código 110: Fibonacci com memoization

Rodando o programa:

```
$ ruby memo2.rb 40
102334155
  0.000095   0.000029   0.000124 (  0.000121)

$ ruby memo2.rb 50
12586269025
```

```
  0.000131   0.000016   0.000147 (  0.000145)

$ ruby memo2.rb 100
354224848179261915075
  0.000199   0.000061   0.000260 (  0.000257)
```

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:

```
require 'benchmark'
require 'memoize'
include Memoize

def fib(numero)
  return numero if numero < 2
  fib(numero - 1) + fib(numero - 2)
end
memoize(:fib, 'memo.cache')

puts Benchmark.measure { puts fib(ARGV[0].to_i) }
```

Código 111: Fibonacci com memoization e caching

Rodando o programa:

```
$ ruby memo3.rb 100
354224848179261915075
  0.007497   0.000000   0.007497 (  0.012052)

$ ruby memo3.rb 100
354224848179261915075
  0.000043   0.000002   0.000045 (  0.000042)
```

Reparem na diferença entre a primeira e segunda vez que o programa é executado. Isso porque temos um arquivo chamado `memo.cache` agora disponível para ir registrando os valores que foram sendo calculados:

```
$ ls memo.cache
-rw-rw-r-- 1 taq taq 1,2K memo.cache
```

Assim dá para brincar um pouco e chutar o balde:

```
$ ruby memo3.rb 100
354224848179261915075
  0.000038   0.000008   0.000046 (  0.000043)

$ ruby memo3.rb 200
280571172992510140037611932413038677189525
  0.013197   0.001269   0.014466 (  0.021126)

$ ruby memo3.rb 350
6254449428820551641549772190170184190608177514674331726439961915653414425
  0.027056   0.011852   0.038908 (  0.051144)
```

Dica

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

```
require 'benchmark'

fib = Hash.new { |h, n| n < 2 ? h[n] = n : h[n] = h[n - 1] + h[n - 2] }

puts Benchmark.measure { puts fib[10] }
55
  0.000036   0.000006   0.000042 (  0.000028)

puts Benchmark.measure { puts fib[100] }
354224848179261915075
  0.000163   0.000030   0.000193 (  0.000184)
```

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:

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

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

Capítulo 7

Threads

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

```
thread = Thread.new do
  puts "Thread #{self.object_id} iniciada!"

  5.times do |valor|
    puts valor
    sleep 1
  end
end

puts "Já criei a thread"
thread.join
```

Código 112: Criando uma thread

Rodando o programa:

```
$ ruby thr1.rb
Thread 84077870 iniciada!
0
Já criei a thread
1
2
3
4
```

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

```
thread = Thread.new do
  puts "Thread #{self.object_id} iniciada!"

  5.times do |valor|
    puts valor
    sleep 1
  end
end

puts "Já criei a thread"
thread.join(3)
```

Código 113: Criando uma thread com timeout

Rodando o programa:

```
$ ruby thr2.rb
Já criei a thread
Thread 76000560 iniciada!
0
1
2
```

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

```
proc = Proc.new do |parametro|
  parametro.times do |valor|
    print "[#{valor + 1}/#{parametro}]"
    sleep 0.5
  end
end

thread = nil
5.times do |valor|
  thread = Thread.new(valor, &proc)
end

thread.join
puts "Terminado!"
```

Código 114: Criando uma thread através de uma Proc

Rodando o programa:

```
$ ruby thr3.rb
[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 em versões antigas da linguagem, como:

```
maior, menor = 0, 0
log = 0

t1 = Thread.new do
  loop do
    maior += 1
    menor -= 1
  end
end

t2 = Thread.new do
  loop do
    log = menor + maior
  end
end

sleep 3
puts "log vale #{log}"
```

Código 115: Falta de sincronia em threads

Rodando o programa:

```
$ rvm 1.8.7
$ ruby thr4.rb
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. Lógico que não vamos ficar utilizando versões antigas da linguagem, mas temos que aprender o que podemos fazer quando tivermos essa falta de sincronia em alguma situação em versões recentes.

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

```
maior, menor = 0, 0
log = 0

mutex = Mutex.new
t1 = Thread.new do
  loop do
    mutex.synchronize do
      maior += 1
      menor -= 1
    end
  end
end

t2 = Thread.new do
  loop do
    mutex.synchronize do
      log = menor+maior
    end
  end
end

sleep 3
puts "log vale #{log}"
```

Código 116: Usando Mutex

Rodando o programa:

```
$ ruby thr5.rb
log vale 0
```

Agora correu tudo como esperado. Podemos alcançar esse resultado também usando `Monitor`:

```
require "monitor"

maior, menor = 0, 0
log = 0
mutex = Monitor.new
t1 = Thread.new do
  loop do
    mutex.synchronize do
      maior += 1
      menor -= 1
    end
  end
end

t2 = Thread.new do
  loop do
    mutex.synchronize do
      log = menor+maior
    end
  end
end

sleep 3
puts "log vale #{log}"
```

Código 117: Usando Monitor

Rodando o programa:

```
$ ruby thr6.rb
log vale 0
```

A diferença dos monitores é que eles podem ser uma classe pai da classe corrente, um `mixin` ou uma extensão de um objeto em particular. Isso permite que possamos escolher o comportamento mais adequado para a situação de uso:

```ruby
require 'monitor'

class Contador1
  attr_reader :valor
  include MonitorMixin

  def initialize
    @valor = 0
    super
  end

  def incrementa
    synchronize do
      @valor = valor + 1
    end
  end
end

c1 = Contador1.new

t1 = Thread.new { 100_000.times { c1.incrementa } }
t2 = Thread.new { 100_000.times { c1.incrementa } }

t1.join
t2.join
puts c1.valor
```

Código 118: Usando Monitor com mixin

```ruby
require 'monitor'

class Contador2
  attr_reader :valor

  def initialize
    @valor = 0
  end

  def incrementa
    @valor = valor + 1
  end
end

c2 = Contador2.new
c2.extend(MonitorMixin)

t3 = Thread.new { 100_000.times { c2.synchronize { c2.incrementa } } }
t4 = Thread.new { 100_000.times { c2.synchronize { c2.incrementa } } }

t3.join
t4.join
puts c2.valor
```

Código 119: Usando Monitor com extend

Rodando:

```
$ ruby thr71.rb
200000
$ ruby thr72.rb
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:

```ruby
require "thread"

items   = []
lock    = Mutex.new
cond    = ConditionVariable.new
limit   = 0

produtor = Thread.new do
  loop do
    lock.synchronize do
      qtde = rand(50)
      next if qtde == 0

      puts "produzindo #{qtde} item(s)"
      items = Array.new(qtde,"item")
      cond.wait(lock)
      puts "consumo efetuado!"
      puts "-" * 25
      limit += 1
    end
    break if limit > 5
  end
end

consumidor = Thread.new do
  loop do
    lock.synchronize do
      if items.length>0
        puts "consumindo #{items.length} item(s)"
        items = []
      end
      cond.signal
    end
  end
end
produtor.join
```

Código 120: Variáveis de condição

Rodando o programa:

```
$ ruby thr8.rb
produzindo 48 item(s)
consumindo 48 item(s)
consumo efetuado!
-------------------------
produzindo 43 item(s)
consumindo 43 item(s)
consumo efetuado!
-------------------------
produzindo 21 item(s)
consumindo 21 item(s)
consumo efetuado!
-------------------------
produzindo 29 item(s)
consumindo 29 item(s)
consumo efetuado!
-------------------------
produzindo 31 item(s)
consumindo 31 item(s)
consumo efetuado!
```

```
------------------------
produzindo 43 item(s)
consumindo 43 item(s)
consumo efetuado!
------------------------
```

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

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

```
require "thread"

queue = Queue.new
limit = 0

produtor = Thread.new do
  loop do
    qtde = rand(50)
    next if qtde.zero?

    limit += 1

    if limit > 5
      queue.enq :END_OF_WORK
      break
    end

    puts "produzindo #{qtde} item(s)"
    queue.enq Array.new(qtde, 'item')
    sleep 1
  end
end

consumidor = Thread.new do
  loop do
    obj = queue.deq
    break if obj == :END_OF_WORK

    print "consumindo #{obj.size} item(s)\n"
  end
end

produtor.join
consumidor.join
```

Código 121: Utilizando Queues

Rodando o programa:

```
$ ruby thr9.rb
produzindo 23 item(s)
consumindo 23 item(s)
produzindo 39 item(s)
consumindo 39 item(s)
produzindo 33 item(s)
consumindo 33 item(s)
produzindo 26 item(s)
consumindo 26 item(s)
produzindo 19 item(s)
consumindo 19 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 [1] (o do exemplo pode ser criado em um sistema Unix-like com `mkfifo teste.fifo`) para criar o bloqueio:

```
proc = Proc.new do |numero|
  loop do
    puts "Proc #{numero}: #{`date`}"
    sleep 3
  end
end

fifo = Proc.new do
  loop do
    puts File.read('teste.fifo')
    sleep 3
  end
end

threads = []

(1..5).each do |numero|
  threads << (numero == 3 ? Thread.new(&fifo) : Thread.new(numero, &proc))
end

threads.each(&:join)
```

Código 122: Utilizando um FIFO

Ali agora com as versões mais recentes tive que introduzir até um `sleep` para que possamos ver o resultado da escrita no `FIFO`, pois roda muito rápido. Podemos fazer algo como `echo 'teste' > teste.fifo` para ver o resultado aparecendo na tela, e ver as threads rodando todas em paralelo. Se quisermos ver o bicho pegar mesmo, podemos remover o `sleep`, mas duvido que vamos ver o resultado da leitura do `FIFO` na tela, de tão rápido que vai ser!

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`:

[1] http://pt.wikipedia.org/wiki/FIFO

```
mutex       = Mutex.new
last_result = 1
last_update = Time.now

trap('SIGINT') do
  puts 'Saindo do programa ...'
  exit
end

Thread.new do
  loop do
    sleep 5
    puts 'Atualizando ...'

    mutex.synchronize do
      # fazendo alguma coisa demorada aqui
      puts 'Mutex sincronizado, vou fazer algo ...'
      sleep 10
      puts 'Terminei de fazer algo no mutex, vou liberar a sincronização'
      last_result += 1
    end

    last_update = Time.now
    puts 'Liberado o mutex.'
  end
end

loop do
  puts 'Aperte ENTER para ver o resultado:'
  gets

  # tenta adquirir o lock, se não estiver entre o synchronize e o end
  # lá em cima
  if mutex.try_lock
    begin
      puts "Resultado atualizado #{(Time.now - last_update).to_i} segundos atrás."
    ensure
      mutex.unlock
    end
  # se não conseguiu, é que está lá em cima, processando entre
  # o synchronize e o end!
  else
    puts "Sendo atualizado, resultado anterior gerado #{(Time.now - last_update).to_i} segundos atrás"
  end
end
```

Código 123: Utilizando try_lock

Rodando o programa:

```
Aperte ENTER para ver o resultado:
Atualizando ...
Mutex sincronizado, vou fazer algo ...
Terminei de fazer algo no mutex, vou liberar a sincronização
Liberado o mutex.

Resultado atualizado 2 segundos atrás.
Aperte ENTER para ver o resultado:
Atualizando ...
Mutex sincronizado, vou fazer algo ...
```

```
Sendo atualizado, resultado anterior gerado 7 segundos atrás
Aperte ENTER para ver o resultado:
```

7.1 Fibers

Entre as *features* introduzidas na versão 1.9, existe uma bem interessante chamada `Fibers`, volta e meia definidas como "*threads* leves". Vamos dar uma olhada nesse código:

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

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

```
enum1 = 3.times
enum2 = %w(zero um dois).each
puts enum1.class

loop do
  puts enum1.next
  puts enum2.next
end
```

Código 124: Enumerators utilizam Fibers

Rodando o programa:

```
Enumerator
0
zero
1
um
2
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:

```
fib = Fiber.new do
  x, y = 0, 1

  loop do
    # envia o valor de y para resume
    # puts 'Indo para o resume'
    Fiber.yield y

    # volta aqui depois do resume
    # puts 'Voltei do resume'
    x, y = y, x + y
  end
end

10.times { puts fib.resume }
```

Código 125: Fibonacci com Fibers

Rodando o programa:

```
1
1
2
3
5
8
13
21
34
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 (se quisermos podemos remover as linhas comentadas com `#` antes de `puts` para ver certinho como está o fluxo do processamento):

- A `Fiber` é criada com `new`.
- Dentro de um iterador que vai rodar 10 vezes, é chamado o método `resume`.
- É executado o código do início do "corpo"da `Fiber` até `yield`.
- Nesse ponto, o controle é transferido com o valor de `y` para onde foi chamado o `resume`, e impresso na tela.
- 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`.
- Pudemos comprovar que `x` e `y` tiveram seus valores preservados entre as trocas de controle.

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

```
def create_fib
  x, y = 0, 1

  lambda do
    t, x, y = y, y, x + y
    return t
  end
end

proc = create_fib
10.times { puts proc.call }
```

Código 126: Usando uma Proc para simular uma Fiber

O resultado ao rodar o programa é o mesmo e 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:

```
f2 = Fiber.new do |value|
  puts "Estou em f2 com #{value}, transferindo para onde vai resumir ..."
  Fiber.yield value + 40
  puts "Cheguei aqui?"
end

f1 = Fiber.new do
  puts "Comecei f1, transferindo para f2 ..."
  f2.resume 10
end

puts "Resumindo fiber 1: #{f1.resume}"
```

Código 127: Comportamento de semirotinas

Rodando o programa:

```
$ ruby fibers4.rb
Comecei f1, transferindo para f2 ...
Estou em f2 com 10, transferindo para onde vai resumir ...
Resumindo fiber 1: 50
```

Comportamento parecido com as semi-corrotinas! Mas e se fizermos isso, transferindo o controle:

```
require "fiber"

f1 = Fiber.new do |other|
  puts "Comecei f1, transferindo para f2 ..."
  other.transfer Fiber.current, 10
end

f2 = Fiber.new do |caller, value|
  puts "Estou em f2, transferindo para f1 ..."
  caller.transfer value + 40
  puts "Cheguei aqui?"
end

puts "Resumindo fiber 1: #{f1.resume(f2)}"
```

Código 128: Transferindo controle entre Fibers

Discussões teóricas à parte, as `Fibers` são um recurso muito interessante. Vamos fazer um esquema de "produtor-consumidor"usando `Fibers`:

```ruby
require "fiber"

produtor = Fiber.new do |cons|
  5.times do
    items = Array.new((rand * 5).to_i + 1, "oi!")
    puts "Produzidos #{items} ..."
    cons.transfer Fiber.current, items
  end
end

consumidor = Fiber.new do |prod, items|
  loop do
    puts "Consumidos #{items}"
    prod, items = prod.transfer
  end
end

produtor.resume consumidor
```

Código 129: Produtor-consumidor com Fibers

Rodando o programa:

```
$ ruby fibers6.rb
Produzidos ["oi!","oi!","oi!","oi!","oi!"] ...
Consumidos ["oi!","oi!","oi!","oi!","oi!"]

Produzidos ["oi!","oi!","oi!","oi!","oi!"] ...
Consumidos ["oi!","oi!","oi!","oi!","oi!"]

Produzidos ["oi!"] ...
Consumidos ["oi!"]

Produzidos ["oi!","oi!","oi!", "oi!", "oi!"] ...
Consumidos ["oi!","oi!","oi!", "oi!", "oi!"]

Produzidos ["oi!","oi!","oi!"] ...
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:

```
str =<<FIM
texto para mostrar como podemos separar palavras do texto
para estatística de quantas vezes as palavras se repetem no
texto
FIM

scanner = Fiber.new do
  str.scan(/\w\p{Latin}+/) do |word|
    Fiber.yield word.downcase
  end
  puts "acabou!"
end

words = Hash.new(0)

while word = scanner.resume
  words[word] += 1
end

words.each do |word,count|
  puts "#{word}:#{count}"
end
```

Código 130: Contando palavras com Fibers

Rodando o programa:

```
$ ruby fibers7.rb
acabou!
texto:3
para:2
mostrar:1
como:1
podemos:1
separar:1
palavras:2
do:1
estatística:1
de:1
quantas:1
vezes:1
as:1
se:1
repetem:1
no:1
```

Dica

Estão vendo como eu escrevi a expressão regular acima? O \p seguido de {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 de Ruby.

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`.

7.1.1 Continuations

Ruby também tem suporte à `continuations`, que são, segundo a continuations:

> Representações abstratas do controle de estado de um programa

Mas, *importante*, está marcado como *deprecated* desde as versões `2.2.x` e deve ser substituído pelas `Fibers`, vistas anteriormente.

Para vermos como elas se comportavam, vamos ver um exemplo que nos mostra que a *call stack* de um programa é preservada chamando uma `Continuation`:

```
require 'continuation'

def cria_continuation
  puts 'Criando a continuation e retornando ...'
  callcc { |obj| return obj }
  puts 'Ei, olha eu aqui de volta na continuation!'
end

puts 'Vou criar a continuation.'
cont = cria_continuation()
puts 'Verificando se existe ...'

if cont
  puts 'Criada, vamos voltar para ela?'
  cont.call
else
  puts 'Agora vamos embora.'
end

puts 'Terminei, tchau.'
```

Código 131: Continuations

Rodando o programa em uma versão como a `2.1.2`:

```
$ ruby cont.rb
Vou criar a continuation.
Criando a continuation e retornando ...
Verificando se existe ...
Criada, vamos voltar para ela?
Ei, olha eu aqui de volta na continuation!
Verificando se existe ...
Agora vamos embora.
Terminei, tchau.
```

Vejam como é definido o processo:

- A `continuation` é criada, retornando para o processo que a criou através de `callcc`.
- Como ela foi criada, é chamado o ponto logo após `callcc` com `call`.
- Com isso, é apresentada a mensagem "olha eu aqui de volta".
- Reparem que voltamos novamente no ponto onde chamamos `cria_continuation`, mas agora ela já foi executada.
- Como foi executada, são mostradas as mensagens de finalização.

7.2 Processos em paralelo

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

```
$ gem install parallel
```

Dica

Para descobrir o número de processadores do computador, podemos utilizar a classe `Etc`:

```
require 'etc'

puts Etc.nprocessors
```

Esse módulo fornece informações armazenadas no diretório `/etc` em sistemas de arquivos `Unix`. O resultado de `nprocessors` pode ser menor que o número de processadores físicos do computador se o processo da VM de Ruby estiver associado à CPUs específicas, retornando os *cores* lógicos, não os físicos.

Vamos ver um exemplo no meu computador corrente, que tem 4 *cores*. Vamos utilizar o método *map* da *gem* `Parallel` para criar um `Array` de 20 elementos de múltiplos de 5, esperando 1 segundo a cada vez que um número for transformado. Se temos uma coleção de 20 elementos onde cada um leva 1 segundo para ser "calculado", então vamos levar 20 segundos para fazer isso, correto? Mas levando em conta que temos 4 *cores* no computador, podemos esperar outra coisa:

```
require 'benchmark'
require 'parallel'

time = Benchmark.measure do
  numbers = Parallel.map 1..20, progress: 'Processando ...' do |num|
    sleep 1
    num * 5
  end
  p numbers
end

puts time
```

Código 132: Utilizando todos os processadores

Rodando o programa:

```
$ ruby parsample.rb
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100]
  0.000000   0.009747   0.037475 (  5.015960)
```

Uia.

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

```ruby
require 'benchmark'
require 'parallel'

time = Benchmark.measure do
  res = 'Quem terminou primeiro? '

  Parallel.map 1..20, in_threads: 4 do |nr|
    puts "Acabei com #{nr} "
    res += "#{nr} "
  end

  puts res
end

puts time
```

Código 133: Parallel com threads

Rodando o programa, vamos ter algo como:

```
$ ruby par.rb
Acabei com 1
Acabei com 2
Acabei com 3
Acabei com 4
Acabei com 8
Acabei com 9
Acabei com 10
Acabei com 11
Acabei com 12
Acabei com 13
Acabei com 14
Acabei com 15
Acabei com 16
Acabei com 17
Acabei com 18
Acabei com 19
Acabei com 20
Acabei com 6
Acabei com 5
Acabei com 7
Quem terminou primeiro? 1 2 3 4 8 9 10 11 12 13 14 15 16 17 18 19 20 6 5 7
  0.002022   0.000532   0.002554 (  0.002090)
```

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`:

```ruby
require 'benchmark'
require 'parallel'

time = Benchmark.measure do
  res = 'Quem terminou primeiro? '

  Parallel.map 1..20, in_processes: 4 do |nr|
    puts "Acabei com #{nr} "
    res += "#{nr} "
  end

  puts res
end

puts time
```

Código 134: Parallel com processos

Rodando o programa:

```
$ ruby par2.rb
Acabei com 1
Acabei com 2
Acabei com 3
Acabei com 5
Acabei com 4
Acabei com 6
Acabei com 7
Acabei com 8
Acabei com 9
Acabei com 10
Acabei com 11
Acabei com 12
Acabei com 13
Acabei com 14
Acabei com 15
Acabei com 16
Acabei com 17
Acabei com 18
Acabei com 19
Acabei com 20
Quem terminou primeiro?
  0.004806   0.000341   0.012969 (  0.006712)
```

Desafio 4

Tente descobrir a diferença entre o código que utilizou `threads` e `processes`.

Dica

Reparem que a frase "Quem terminou primeiro?"veio sem mais nada no final, e reparem também que estamos utilizando agora **processos** e que eu disse que `Threads` permitem a modificação de dados globais e processos não.

Para executar esse mesmo código utilizando o número de processadores da CPU, é só não especificar nem `in_threads` ou `in_processes`, similar ao primeiro evento que vimos:

```ruby
require 'benchmark'
require 'parallel'

time = Benchmark.measure do
  res = 'Quem terminou primeiro? '

  Parallel.map 1..20 do |nr|
    puts "Acabei com #{nr} "
    res += "#{nr} "
  end

  puts res
end

puts time
```

Código 135: Parallel utilizando o número de processadores locais

Rodando o programa:

```
± ruby par3.rb
Acabei com 1
Acabei com 2
Acabei com 3
Acabei com 5
Acabei com 4
Acabei com 7
Acabei com 8
Acabei com 6
Acabei com 9
Acabei com 10
Acabei com 11
Acabei com 12
Acabei com 13
Acabei com 14
Acabei com 15
Acabei com 16
Acabei com 17
Acabei com 18
Acabei com 19
Acabei com 20
Quem terminou primeiro?
  0.000935   0.004218   0.013757 (  0.007475)
```

Fazendo uma comparação com Threads:

```ruby
require 'benchmark'

threads = []

time = Benchmark.measure do
  res = 'Quem terminou primeiro? '

  (1..20).each do |nr|
    threads << Thread.new do
      puts "acabei com #{nr} "
      res += "#{nr} "
    end
  end

  threads.each(&:join)
  puts res
end

puts time
```

Código 136: Comparando Parallel com threads

```
$ ruby par4.rb
acabei com 2
acabei com 1
acabei com 3
acabei com 4
acabei com 5
acabei com 7
acabei com 6
acabei com 8
acabei com 10
acabei com 9
acabei com 11
acabei com 12
acabei com 13
acabei com 14
acabei com 15
acabei com 17
acabei com 16
acabei com 18
acabei com 19
acabei com 20
Quem terminou primeiro? 2 1 3 4 5 7 6 8 10 9 11 12 13 14 15 17 16 18 19 20
  0.004020   0.004909   0.008929 (  0.007937)
```

Isso nos deu os seguintes resultados (que, lógico, vão depender do computador que você está rodando o código):

Parallel com threads	0.002090
Parallel com processos	0.006712
Parallel com processadores locais	0.007475
Threads	0.007937

O método a ser utilizado depende dos fatores de isolamento do código, sincronização, etc. Podemos analisar para ver qual seria o melhor. Como dica, experimentem no arquivo `par.rb` utilizar `in_threads: 1` e vejam como fica o resultado.

7.3 Benchmarks

Ao invés de medir nosso código através do sucessivas chamadas à `Time.now`, podemos utilizar o módulo de *benchmark*, como visto nos exemplos anteriores. Vamos fazer uma nova medição aqui, primeiro medindo uma operação simples, como criar uma `String` enorme, executando o seguinte código dentro do `irb`:

```
require "benchmark"

puts Benchmark.measure { "-" * 1_000_000 }
0.000500   0.000107   0.000607 (  0.000606)
```

Ou medir um pedaço de código:

```
require 'benchmark'
require 'parallel'

Benchmark.bm do |bm|
  bm.report do
    Parallel.map 1..20, in_threads: 4 do |nr|
      5.times { |t| sleep rand }
    end
  end
end
```

Código 137: Fazendo benchmark de um pedaço de código

```
$ ruby bench1.rb
       user     system      total        real
   0.007754   0.002154   0.009908 ( 13.575916)
```

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

```
require 'benchmark'
require 'parallel'

Benchmark.bm do |bm|
  bm.report('in_threads') do
    Parallel.map 1..20, in_threads: 4 do |nr|
      5.times { |t| sleep 0.5 }
    end
  end

  bm.report('in_processes') do
    Parallel.map 1..20, in_processes: 4 do |nr|
      5.times { |t| sleep 0.5 }
    end
  end

  bm.report('using threads') do
    threads = []

    (1..20).each do |nr|
      threads << Thread.new do
        5.times { |t| sleep 0.5 }
      end
    end
    threads.each(&:join)
  end
end
```

Código 138: Fazendo benchmark de pedaços de código

Rodando o programa:

```
$ ruby bench2.rb
                 user     system      total        real
in_threads    0.000000   0.007713   0.007713 ( 12.505493)
```

```
in_processes  0.009806   0.014331   0.067245 ( 12.551572)
using threads 0.008777   0.033563   0.042340 (  2.554518)
```

7.4 Ractors

A partir da versão `3.0.0`, temos disponível a classe `Ractor` (chamados de `Guild` anteriormente), que apesar de ter começado como experimental, despertou grande interesse da comunidade dos desenvolvedores.

Vimos acima que temos algumas situações onde temos situações em que o código roda em *paralelo*, onde várias tarefas podem começar e terminar de forma simultânea, às vezes com situações *concorrentes*, onde só um recurso está sendo processado em determinado momento, especialmente se estiverem disputando um mesmo recurso.

Antes da versão `3.0.0`, a linguagem Ruby por si só não suportava processos paralelos verdadeiros, por causa do GIL (Global Interpreter Lock), que age como um semáforo fazendo a distribuição entre os processos, fazendo com que o código não seja rodado de maneira simultânea.

Inclusive, para permitir que os códigos entre as `Threads` sejam executados de forma paralela e segura, *cada* `Ractor` tem o seu próprio GIL e o seu próprio contexto onde roda o código, mantendo isolados os objetos dos outros `Ractors`, onde se comunicam através de um protocolo de comunicação que utiliza duas abordagens. Como curiosidade, o nome `Ractor` vem de "actor model".

Os objetos que são compartilháveis através dessas abordagens são chamados *"shareable"*, que são: `Integer`, `Strings` e outros objetos imutáveis ("congelados") e o próprio objeto do `Ractor`. Podemos testar se um objeto é *"shareable"* através do método *shareable?*:

```
Ractor.shareable?(1)          # true
Ractor.shareable?([])         # false
Ractor.shareable?([].freeze)  # true
```

Quando o objeto não é *"shareable"* é feita um *deep copy* do objeto, porém alguns objetos não estão aptos para terem isso implementado para o uso nos `Ractors` (como `Threads`, então fiquem de olho!

Vamos dar uma olhada nas abordagens de comunicação com os `Ractors`.

A primeira abordagem é através de `send/receive`, que é do tipo *"push type"*, onde através de `send` enviamos um objeto para ser coletado por `receive` dentro do `Ractor`. Nessa abordagem, como o tipo diz, estamos *enviando informação para dentro* do `Ractor`. Um exemplo disso:

```
ractor = Ractor.new do
  puts "Recebida mensagem: #{Ractor.receive}"
end

ractor.send "Olá, mundo dos Ractors!"
sleep 1
```

Código 139: Ractor com send/receive

Rodando o programa (prestem atenção na `flag`):

```
$ ruby -W0 rac1.rb
Recebida mensagem: Olá, mundo dos Ractors!
```

Algumas coisas a serem notadas aqui:

- Utilizamos a flag `-W0` para desabilitar uma mensagem de *warning* que recebemos pelo fato dos `Ractors` serem um recurso experimental.
- Utilizamos `sleep` para aguardar 1 segundo antes do interpretador sair do código. Se não fizermos isso, ele vai terminar antes de obtermos o resultado, ou seja, a mensagem recebida e impressa em `receive`!

Podemos utilizar a flag `-W` seguida de um número para indicar ao interpretador da linguagem qual o nível de alerta (*warning*) que desejamos:

- 0 - silêncio completo
- 1 - nível médio
- 2 - nível verboso

A segunda abordagem é através de `yield/take`, que é do tipo *"pull type"*, onde *recebemos informações* de dentro do `Ractor`. Um exemplo disso:

```
ractor = Ractor.new do
  Ractor.yield Time.now.strftime('%H:%M')
end

puts "A hora corrente no mundo do Ractor é #{ractor.take}"
sleep 1
```

Código 140: Ractor com yield/take

Rodando o programa, vamos ter algo como:

```
$ ruby -W0 rac2.rb
A hora corrente no mundo do Ractor é 15:21
```

Vamos dar uma olhada em como ficaria algo como "produtor/consumidor"utilizando `Ractors`. Reparem que o argumento passado em `new` se encaixa no bloco do `Ractor` (nesse caso, ambos com nome de `consumer`, mas um é a referência *externa* que vira a referência *interna*):

```
consumer = Ractor.new do
  loop do
    items = Ractor.receive
    puts "Recebidos #{items} itens."
  end
end

producer = Ractor.new(consumer) do |consumer|
  5.times do |num|
    items = Array.new(num, 'item')
    puts "Enviando #{items.size}\n"
    consumer.send items.size
    sleep 0.1
  end
end

sleep 5
```

Código 141: Produtor/consumidor com Ractors

Rodando o programa:

```
$ ruby -W0 rac3.rb
Enviando 0
Recebidos 0 itens.
Enviando 1
Recebidos 1 itens.
Enviando 2
Recebidos 2 itens.
Enviando 3
Recebidos 3 itens.
Enviando 4
Recebidos 4 itens.
```

Vejam que foi utilizado o `sleep` novamente, para dar tempo do consumidor ser notificado e aguardar no final do programa. Se removermos, vai ficar com um comportamento basicamente síncrono, pois o loop ali vai ser executado direto e o interpretador é capaz de terminar mais rápido que algum *output*.

Mas e se quisermos enviar o `Array` e não apenas o inteiro mostrando o tamanho dele? Nesse caso, vamos enviar um `Array` imutável, congelado:

```
consumer = Ractor.new do
  loop do
    items = Ractor.receive
    puts "Recebidos #{items.size} itens."
    p items
  end
end

producer = Ractor.new(consumer) do |consumer|
  5.times do |num|
    items = Array.new(num, 'item').freeze
    puts "Enviando #{items}\n"
    consumer.send items
    sleep 0.1
  end
end

sleep 5
```

Código 142: Produtor/consumidor com Ractors e Array imutável

Rodando o programa:

```
$ ruby -W0 rac4.rb
Enviando []
Recebidos 0 itens.
[]
Enviando ["item"]
Recebidos 1 itens.
["item"]
Enviando ["item", "item"]
Recebidos 2 itens.
["item", "item"]
Enviando ["item", "item", "item"]
Recebidos 3 itens.
["item", "item", "item"]
Enviando ["item", "item", "item", "item"]
Recebidos 4 itens.
["item", "item", "item", "item"]
```

Para provar que o bloco do `Ractor` está isolado do resto, podemos tentar acessar uma variável de fora, o que vai nos dar uma exceção do tipo `ArgumentError`:

```
nome = "taq"
ractor = Ractor.new do
  Ractor.yield nome # para funcionar, troquem nome para "taq"
end

puts "o nome é #{ractor.take}"
```

Código 143: Erro no Ractor tentando acessar fora do bloco

Rodando o programa:

```
$ ruby -W0 rac5.rb
```

```
<internal:ractor>:267:in `new': can not isolate a Proc because it accesses outer variables (nome). (ArgumentE
        from rac5.rb:2:in `<main>'
```

Vamos tirar aquele `sleep` final do produtor/consumidor? Para isso, vamos utilizar o método `take`:

```
consumer = Ractor.new do
  loop do
    items = Ractor.receive
    puts "Recebidos #{items.size} itens."
    p items
  end
end

producer = Ractor.new(consumer) do |consumer|
  5.times do |num|
    items = Array.new(num, 'item').freeze
    puts "Enviando #{items}\n"
    consumer.send items
    sleep 0.1
  end
end

producer.take
```

Código 144: Removendo o sleep final do produtor/consumidor

Rodando o programa:

```
$ ruby -W0 rac6.rb
Enviando []
Recebidos 0 itens.
[]
Enviando ["item"]
Recebidos 1 itens.
["item"]
Enviando ["item", "item"]
Recebidos 2 itens.
["item", "item"]
Enviando ["item", "item", "item"]
Recebidos 3 itens.
["item", "item", "item"]
Enviando ["item", "item", "item", "item"]
Recebidos 4 itens.
["item", "item", "item", "item"]
```

E se quisermos esperar vários `Ractors`? Nesse caso, podemos utilizar o método `select`, tomando o cuidado para receber os `Ractors` que terminam e eliminar do `Array`:

```ruby
consumer = Ractor.new do
  loop do
    items = Ractor.receive
    puts "Recebidos #{items.size} itens."
    p items
  end
end

p1 = Ractor.new(consumer) do |consumer|
  5.times do |num|
    items = Array.new(num, 'item').freeze
    puts "Enviando #{items}\n"
    consumer.send items
    sleep 0.1
  end
end

p2 = Ractor.new(consumer) do |consumer|
  5.times do |num|
    items = Array.new(num, 'new item').freeze
    puts "Enviando #{items}\n"
    consumer.send items
    sleep 0.1
  end
end

producers = [p1, p2]
producers.size.times do
  r, n = Ractor.select *producers
  producers.delete(r)
end
```

Código 145: Esperando vários Ractors terminarem

Rodando o programa:

```
$ ruby -W0 rac7.rb
Enviando []
Recebidos 0 itens.
[]
Enviando []
Recebidos 0 itens.
[]
Enviando ["item"]
Enviando ["new item"]
Recebidos 1 itens.
["item"]
Recebidos 1 itens.
["new item"]
Enviando ["new item", "new item"]
Enviando ["item", "item"]
Recebidos 2 itens.
["item", "item"]
Recebidos 2 itens.
["new item", "new item"]
Enviando ["item", "item", "item"]
Enviando ["new item", "new item", "new item"]
Recebidos 3 itens.
["new item", "new item", "new item"]
Recebidos 3 itens.
["item", "item", "item"]
```

```
Enviando ["new item", "new item", "new item", "new item"]
Enviando ["item", "item", "item", "item"]
Recebidos 4 itens.
["new item", "new item", "new item", "new item"]
Recebidos 4 itens.
["item", "item", "item", "item"]
```

Um comportamento interessante é que podemos mover referências para um `Ractor`, com o detalhe importante de que *a referência local ficará inválida*, não permitindo enviar nenhuma mensagem para ela, disparando uma exceção do tipo `Ractor::MovedError` se tentarmos:

```
ractor = Ractor.new do
  obj = Ractor.receive
  obj << "world"
end

obj = ["hello"]
ractor.send obj, move: true

modificado = ractor.take
puts modificado.join(" ")
puts obj.join(" ")
```

Código 146: Movendo objetos para o Ractor

Rodando o programa:

```
$ ruby -W0 rac8.rb
hello world
rac8.rb:11:in `method_missing': can not send any methods to a moved object (Ractor::MovedError)
        from rac8.rb:11:in `<main>'
```

Vale lembrar que os `Ractors` ainda são recursos experimentais, então usem com sabedoria!

Capítulo 8

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.

8.1 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:

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

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

```
p File.read("teste.txt")
```

Código 147: Lendo um arquivo texto

Rodando o programa:

```
$ ruby io1.rb
"Arquivo de teste\nCurso de Ruby\nEstamos na terceira linha.\nE aqui é a quarta 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`):

```
p File.readlines("teste.txt")
```

Código 148: Lendo um arquivo texto com quebras de linhas

Rodando o programa:

```
$ ruby io2.rb
["Arquivo de teste\n", "Curso de Ruby\n", "Estamos na terceira linha.\n", "E aqui é a quarta e última.\n"]
```

Cuidado ao ler arquivos muito grandes dessa forma! Será gerado um `Array` com o número de elementos igual ao número de linhas do arquivo e com o quantidade de caracteres de cada linha!

> "FileReader ... FileInputStream ... BufferedReader ... como que era mesmo?"Vai falar, você aí que programa em Java, apesar de não serem fluxos especializados, não é bem mais prático por aqui?

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:

```
f = File.open("teste.txt")

f.each do |linha|
  puts linha
end

f.close
```

Código 149: Lendo um arquivo texto linha a linha

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

```
File.open("teste.txt") do |arquivo|
  arquivo.each do |linha|
    puts linha
  end
end
```

Código 150: Fechando um arquivo automagicamente

Rodando o programa, é o mesmo resultado acima, com a diferença que isso "automagicamente"vai fechar o *handle* do arquivo, no final do bloco. Confessa aí, você já deixou um handle de arquivo, conexão com o banco, conexão de rede aberta alguma vez, né não? Esse é um pecado infelizmente comum com os desenvolvedores, deixar aberto recursos após eles não serem mais necessários. Isso dá um belo esforço para o *garbage collector* e deixa o código do *runtime* mais poluído desnecessariamente. Não deixem de limpar a sua área de trabalho, ou seja, o seu código!

Lendo dados no mesmo arquivo

Ruby tem um objeto de `IO` chamado `DATA`, que retorna as linhas definidas após `__END__` e o fim do arquivo. Dessa forma, podemos carregar alguns dados junto com nosso código-fonte.

Vamos fazer um pequeno teste com o recurso da dica acima:

```
DATA.each do |linha|
  puts "linha: #{linha}"
end

__END__
Esse é um teste
de dados
embutidos no arquivo
de código-fonte
```

Código 151: Lendo um arquivo texto

Rodando o programa:

```
$ ruby code/io/data.rb
linha: Esse é um teste
linha: de dados
linha: embutidos no arquivo
linha: de código-fonte
```

Vamos aproveitar que estamos lendo arquivos e ver uma implementação do operador `flip-flop`, mencionado anteriormente. Ele memoriza um estado de `true` quando a primeira expressão é avaliada como `true`, e fica nesse estado até a segunda expressão é avaliada também como `true`, quando fica sendo avaliado como `false` até a primeira expressão voltar novamente a ser avaliada como `true`.

Vamos levar em conta que vamos ler cada linha do arquivo, que será armazenada em uma variável chamada `line`. A primeira expressão vai ser

```
line.match?(/^\_start_/)
```

ou seja, vai avaliar se a linha corrente combina com a expressão regular indicada, que tem `_start _`, e a segunda expressão será

```
line.match?(/^_end_/)
```

que é basicamente igual à anterior, com a diferença de que agora temos `_end _`. Para indicarmos que queremos o `flip-flop`, vamos fazer algo parecido com uma `Range`:

```
if line.match?(/^\_start_/) .. line.match?(/^_end_/)
```

ou seja, *quando for encontrada a primeira linha que "case"com start e até que seja encontrada uma que "case"com end*, o corpo do *if* será executado. Do contrário, será o do `else`.

O conteúdo do arquivo é:

```
_start_
Esse é um parágrafo de texto
que vai ser indentado.
_end_
esse não vai ter indentação.
esse também não.
_start_
E aqui tem outro!
_end_
_start_
E mais um aqui
com mais linhas
só de teste.
_end_
```

E o código é:

```
File.open('flip_flop_test.txt').each do |line|
  if line.match?(/^\_start_/) .. line.match?(/^_end_/)
    puts "\t#{line}".gsub(/_\w+_/, '')
  else
    puts line
  end
end
```

Código 152: Lendo um arquivo texto usando flip-flop

Rodando o programa vamos ter o comportamento esperado:

```
$ ruby flip_flop.rb


        Esse é um parágrafo de texto
        que vai ser indentado.

esse não vai ter indentação.
esse também não.

        E aqui tem outro!


        E mais um aqui
```

```
        com mais linhas
        só de teste.
```

Vamos voltar para o nosso arquivo `teste.txt`. Para ler o arquivo *byte* a *byte*, podemos fazer:

```
File.open("teste.txt") do |arquivo|
  arquivo.each_byte do |byte|
    print "[#{byte}]"
  end
end
```

Código 153: Lendo um arquivo byte a byte

Rodando o programa:

```
$ ruby bytes.rb
[65][114][113][117][105][118][111][32][100][101][32][116][101][115][116]
[101][10][67][117][114][115][111][32][100][101][32][82][117][98][121]
[10][69][115][116][97][109][111][115][32][110][97][32][116][101][114]
[99][101][105][114][97][32][108][105][110][104][97][46][10][69][32][97]
[113][117][105][32][195][169][32][97][32][113][117][97][114][116][97]
[32][101][32][195][186][108][116][105][109][97][46][10]
```

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

```
File.open("teste.txt") do |arquivo|
  arquivo.each_char do |char|
    print "[#{char}]"
  end
end
```

Código 154: Lendo um arquivo caracter a caracter

Rodando o programa:

```
[A][r][q][u][i][v][o][ ][d][e][ ][t][e][s][t][e][
][C][u][r][s][o][ ][d][e][ ][R][u][b][y][
][E][s][t][a][m][o][s][ ][n][a][ ][t][e][r][c][e][i][r][a][ ][l][i][n][h][a][.][
][E][ ][a][q][u][i][ ][é][ ][a][ ][q][u][a][r][t][a][ ][e][ ][ú][l][t][i][m][a][.][
]
```

Olhem que moleza fazer uma cópia de um arquivo:

```
File.open("novo_teste.txt","w") do |arquivo|
  arquivo << File.read("teste.txt")
end
```

Código 155: Copiando um arquivo, lendo e escrevendo

8.2 FileUtils

O exemplo anterior foi legalzinho, mas ainda estamos lendo todo o arquivo e escrevendo novamente. Podemos fazer isso de modo mais otimizado utilizando a classe `FileUtils` do Ruby, da seguinte forma:

```
require 'fileutils'

FileUtils.cp 'teste.txt', 'novo_teste.txt'
```

Podemos mover o arquivo:

```
require 'fileutils'

FileUtils.mv 'teste.txt', 'novo_teste.txt'
```

Temos no `FileUtils` vários outros métodos baseados nos comandos dos sistemas de arquivos `Unix`, como por exemplo:

`cd, chdir, chmod, chown, cp, ln, mkdir, mv, pwd, rm, rmdir, touch`

e vários outros. Para ver todos, podemos consultar a documentação *online* desse módulo.

8.3 Arquivos Zip

Podemos ler e escrever em arquivos compactados `Zip`, para isso vamos precisar instalar a *gem* `rubyzip`:

```
$ 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 `.zip` em um subdiretório chamado `txts`, compactando e logo descompactando:

```
require "zip"
require "fileutils"

myzip = "teste.zip"
File.delete(myzip) if File.exists?(myzip)

Zip::File.open(myzip,true) do |zipfile|
  Dir.glob("[0-9]*.txt") do |file|
    puts "Zipando #{file}"
    zipfile.add("txts/#{file}", file)
  end
end

Zip::File.open(myzip) do |zipfile|
  zipfile.each do |file|
    dir = File.dirname(file.name)
    puts "Descompactando #{file.name} para #{dir}"
    FileUtils.mkpath(dir) if !File.exists?(dir)
    zipfile.extract(file.name,file.name) do |entry, file|
      puts "Arquivo #{file} existe, apagando ..."
      File.delete(file)
    end
  end
end
```

Código 156: Criando um arquivo zip

Rodando o programa:

```
$ ruby io7.rb
Zipando 1.txt
Zipando 2.txt
Zipando 3.txt
Descompactando txts/1.txt para txts
Descompactando txts/2.txt para txts
Descompactando txts/3.txt para txts


$ ls txts
total 20K
drwxr-xr-x 2 taq taq .
```

```
drwxr-xr-x 6 taq taq ..
-rw-r--r-- 1 taq taq 1.txt
-rw-r--r-- 1 taq taq 2.txt
-rw-r--r-- 1 taq taq 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.

8.4 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.

Antes de mais nada, vamos criar um arquivo `XML` para os nossos testes, chamado `aluno.xml`, usando o `REXML` para isso:

```ruby
require "rexml/document"

doc  = REXML::Document.new
decl = REXML::XMLDecl.new("1.0", "UTF-8")
doc.add decl

root = REXML::Element.new("alunos")
doc.add_element root

alunos = [
  [1, "João"],
  [2, "José"],
  [3, "Antonio"],
  [4, "Maria"]
]

alunos.each do |info|
  aluno = REXML::Element.new("aluno")
  id    = REXML::Element.new("id")
  nome  = REXML::Element.new("nome")

  id.text   = info[0]
  nome.text = info[1]

  aluno.add_element id
  aluno.add_element nome
  root.add_element aluno
end

doc.write(File.open("alunos.xml", "w"))
```

Código 157: Criando um arquivo XML

Rodando o programa:

```
$ ruby rexml.rb
```

O resultado será algo como:

```
$ cat alunos.xml
<?xml version='1.0' encoding='UTF-8'?>
<alunos>
  <aluno>
      <id>1</id>
      <nome>João</nome>
  </aluno>
  <aluno>
      <id>2</id>
      <nome>José</nome>
  </aluno>
  <aluno>
      <id>3</id>
      <nome>Antonio</nome>
  </aluno>
  <aluno>
      <id>4</id>
      <nome>Maria</nome>
  </aluno>
</alunos>
```

Agora vamos ler esse arquivo. Vamos supor que eu quero listar os dados de todos os alunos:

```
require "rexml/document"

doc = REXML::Document.new(File.open("alunos.xml"))

doc.elements.each("alunos/aluno") do |aluno|
  puts "#{aluno.elements['id'].text} - #{aluno.elements['nome'].text}"
end
```

Código 158: Lendo um arquivo XML

Rodando o programa:

```
$ ruby xml2.rb
1 - João
2 - José
3 - Antonio
4 - Maria
```

Poderíamos ter convertido também os elementos em um `Array` e usado o iterador para percorrer o arquivo, o que dará resultado similar:

```
require "rexml/document"

doc = REXML::Document.new(File.open("alunos.xml"))

doc.elements.to_a("//aluno").each do |aluno|
  puts "#{aluno.elements['id'].text} - #{aluno.elements['nome'].text}"
end
```

Código 159: Lendo um arquivo XML usando arrays

Se quiséssemos somente o segundo aluno, poderíamos usar:

```
require "rexml/document"

doc   = REXML::Document.new(File.open("alunos.xml"))
root  = doc.root
aluno = root.elements["aluno[2]"]

puts "#{aluno.elements['id'].text} - #{aluno.elements['nome'].text}"
```

Código 160: Selecionando um elemento XML pela posição

Rodando o programa:

```
$ ruby xml4.rb
2 - José
```

Uma abordagem mais moderna para criar `XML` em Ruby é a *gem* `builder`:

```
$ gem install builder
```

```ruby
require "builder"

alunos = {
  1 => "João",
  2 => "José",
  3 => "Antonio",
  4 => "Maria"
}

xml = Builder::XmlMarkup.new(indent: 2)
xml.instruct!

xml.alunos do
  alunos.each do |key, value|
    xml.aluno do
      xml.id key
      xml.nome value
    end
  end
end

# para gravar o arquivo
File.open("alunos.xml", "w") do |file|
  file << xml.target!
end
```

Código 161: Criando um arquivo XML com a gem builder

Rodando o programa e verificando o arquivo:

```
$ ruby xml5.rb
$ cat alunos.xml
<alunos>
  <aluno>
     <id>1</id>
     <nome>João</nome>
  </aluno>
...
```

E para a leitura de arquivos `XML`, podemos utilizar a *gem* `nokogiri`:

```
$ gem install nokogiri
```

> Talvez para fazer a instalação da `Nokogiri`, vai ser preciso instalar algumas *libs* no sistema operacional. No Ubuntu, são:
>
> ```
> $ sudo apt-get install libxml2-dev libxslt-dev
> ```
>
> Várias *gems* fazem compilação de código local para utilizar recursos do sistema operacional. É por isso que em sistemas Windows não temos alguns desses recursos.

```ruby
require "nokogiri"

doc = Nokogiri::XML(File.open("alunos.xml"))

doc.search("aluno").each do |node|
  puts node.search("id").text + " - " + node.search("nome").text
end
```

Código 162: Lendo um arquivo XML com Nokogiri

Rodando o programa:

```
$ ruby xml6.rb
1 - João
2 - José
3 - Antonio
4 - Maria
```

8.5 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 [1].

O uso de `XSLT` em Ruby pode ser feito com o uso da *gem* `ruby-xslt`:

> Aqui vamos precisar de uma *lib* do sistema operacional instalada para compilar a *gem*. No `Ubuntu`, ela é a `libxslt-dev`:
>
> ```
> $ sudo apt install libxslt-dev
> ```

```
$ 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] http://www.w3.org/TR/xslt

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html" encoding="utf-8" indent="no"/>
  <xsl:template match="/alunos">
    <html>
      <head>
        <title>Teste de XSLT</title>
      </head>
      <body>
        <table>
          <caption>Alunos</caption>
          <thead>
            <th>
              <td>Id</td>
              <td>Nome</td>
            </th>
          </thead>
          <tbody>
            <xsl:apply-templates/>
          </tbody>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="aluno">
    <tr>
      <td><xsl:value-of select="id"/></td>
      <td><xsl:value-of select="nome"/></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Código 163: Arquivo XSLT

Agora o código Ruby:

```
require "xml/xslt"

xslt = XML::XSLT.new
xslt.xsl = "alunos.xsl"
xslt.xml = "alunos.xml"
xslt.save("alunos.html")
puts xslt.serve
```

Código 164: Processando XSLT

Rodando o programa vamos ter o resultado gravado no arquivo `alunos.html` e apresentado na tela. Abrindo o arquivo com o navegador texto `lynx` (navegador bem legal, se não quisermos conhecer ou tivermos instalado, podemos abrir o arquivo `alunos.html` com qualquer outro navegador, ignorando ali após o *pipe*) vamos ver:

```
$ ruby xslt.rb | lynx --stdin
CAPTION: Alunos
Id Nome
1 João
2 José
3 Antonio
4 Maria
```

8.6 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: JSON[2]. 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:

```
require "json"

{ joao: 1, jose: 2, antonio: 3, maria: 4 }.to_json
=> "{"joao":1,"jose":2,"antonio":3,"maria":4}"
```

e a conversão de volta:

```
JSON.parse({ joao: 1, jose: 2, antonio: 3, maria: 4 }.to_json)
=> {"joao"=>1, "jose"=>2, "antonio"=>3, "maria"=>4}
```

Temos um truque legal para utilizarmos uma `String` em formato JSON, convertendo ela para algo parecido com um "objeto", utilizando nossa conhecida `OpenStruct`:

```
require 'json'

json = { nome: 'Eustáquio', sobrenome: 'Rangel' }.to_json
 => "{\"nome\":\"Eustáquio\",\"sobrenome\":\"Rangel\"}"
obj = OpenStruct.new(JSON.parse(json))
 => #<OpenStruct nome="Eustáquio", sobrenome="Rangel">
obj.nome
 => "Eustáquio"
obj.sobrenome
 => "Rangel"
```

8.7 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`:

```
---
  - josé
  - joão
  - antonio
  - maria
```

Código 165: Arquivo YAML

E agora vamos ler esse arquivo, tendo o resultado convertido em um `Array`:

```
require "yaml"

result = YAML::load(File.open(ARGV[0]))
p result
```

Código 166: Lendo um arquivo YAML

[2] http://www.json.org/

Rodando o programa:

```
$ ruby leryaml.rb teste.yml
["josé", "joão", "antonio", "maria"]
```

Podemos ter `Arrays` dentro de `Arrays`:

```
---
  -
    - joão
    - josé
  -
    - maria
    - antonio
```

Código 167: Arquivo YAML com arrays

Rodando o programa:

```
$ ruby leryaml.rb teste2.yml
[["joão", "josé"], ["maria", "antonio"]]
```

Agora vamos ver como fazer uma `Hash`:

```
---
josé: 1
joão: 2
antonio: 3
maria: 4
```

Código 168: Arquivo YAML com hashes

Rodando o programa:

```
$ ruby leryaml.rb teste3.yml
{"josé"=>1, "joão"=>2, "antonio"=>3, "maria"=>4}
```

`Hashes` dentro de `Hashes`:

```
---
pessoas:
    joão: 1
    josé: 2
    maria: 3
    antonio: 4
```

Código 169: Arquivo YAML com hashes dentro de hashes

Rodando o programa:

```
$ ruby leryaml.rb teste4.yml
{"pessoas"=>{"joão"=>1, "josé"=>2, "maria"=>3, "antonio"=>4}}
```

O que nos dá, com um arquivo de configuração do banco de dados do `Rails`:

```
---
development:
    adapter: mysql
    database: teste_development
    username: root
    password: test
    host: localhost

test:
    adapter: mysql
    database: teste_test
    username: root
    password: test
    host: localhost

production:
    adapter: mysql
    database: teste_production
    username: root
    password: test
    host: localhost
```

Código 170: Arquivo YAML com configurações do Rails

Rodando o programa:

```
$ ruby leryaml.rb teste5.yml
{"development"=>{"adapter"=>"mysql", "database"=>"teste_development",
"username"=>"root", "password"=>"test", "host"=>"localhost"},
"test"=>{"adapter"=>"mysql", "database"=>"teste_test", "username"=>"root",
"password"=>"test", "host"=>"localhost"}, "production"=>{"adapter"=>"mysql",
"database"=>"teste_production", "username"=>"root", "password"=>"test",
"host"=>"localhost"}}
```

8.8 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 impossilibidade 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:

```
require "socket"

TCPSocket.open("smtp.mailtrap.io", 465) do |smtp|
  puts smtp.gets
  smtp.puts "EHLO bluefish.com.br"
  puts smtp.gets
end
```

Código 171: Lendo um socket TCP

Rodando o programa:

```
$ ruby sock.rb
220 mailtrap.io ESMTP ready
```

```
250-mailtrap.io
```

Servidor de email para testes

O serviço mailtrap.io é uma opção bem interessante para brincar um pouco e testar os emails enviados para a sua aplicação.

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)[3]:

```
require "socket"

TCPServer.open("localhost", 8081) do |server|
  puts "servidor iniciado"

  loop do
    puts "aguardando conexão ..."
    con = server.accept
    puts "conexão recebida!"
    con.puts "Sua conexão foi recebida!"
    con.close
  end
end
```

Código 172: Criando um servidor TCP

Rodando o programa:

```
$ ruby tcpserver.rb
servidor iniciado
aguardando conexão ...
conexão recebida!

$ telnet localhost 8081
Trying ::1...
Connected to localhost.localdomain.
Escape character is '^]'.
Sua conexão foi recebida!
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`:

[3] https://gist.github.com/taq/5793430

```ruby
require "socket"

TCPServer.open("localhost", 8081) do |server|
  puts "servidor iniciado"

  loop do
    puts "aguardando conexão ..."
    con = server.accept
    rst = con.recv(1024).unpack("LA10A*")
    fix = rst[0]
    str = rst[1]

    hash = Marshal.load(rst[2])
    puts "#{fix.class}\t: #{fix}"
    puts "#{str.class}\t: #{str}"
    puts "#{hash.class}\t: #{hash}"
    con.close
  end
end
```

Código 173: Servidor TCP com objetos

E agora com o arquivo do cliente, `tcpclient.rb`:

```ruby
require "socket"

hash = { um: 1, dois: 2, tres: 3 }

TCPSocket.open("localhost", 8081) do |server|
  server.write [
    1,
    "teste".ljust(10),
    Marshal.dump(hash)
  ].pack("LA10A*")
end
```

Código 174: Cliente TCP

Abrimos um terminal novo, e rodamos o servidor:

```
$ ruby tcpserver2.rb
servidor iniciado
aguardando conexão ...
```

E agora em outro terminal, rodamos o cliente:

```
$ ruby tcpclient.rb
```

Resultado no servidor:

```
Fixnum  : 1
String  : teste
Hash    : {:um=>1, :dois=>2, :tres=>3}
aguardando conexão ...
```

Desafio 6

Você consegue descobrir o que significa aquele código que foi utilizado no método `pack`?

8.9 UDP

O protocolo UDP[4] 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:

```
require "socket"

server = UDPSocket.new
porta  = 12345
server.bind("localhost", porta)
puts "Servidor conectado na porta #{porta}, aguardando ..."

loop do
  msg, sender = server.recvfrom(256)
  host = sender[3]
  puts "Host #{host} enviou um pacote UDP: #{msg}"
  break unless msg.chomp != "kill"
end

puts "Kill recebido, fechando servidor."
server.close
```

Código 175: Servidor UDP

Agora o código do cliente:

```
require "socket"

client = UDPSocket.open
client.connect("localhost", 12345)

loop do
  puts "Digite sua mensagem (quit termina, kill finaliza servidor):"
  msg = gets
  client.send(msg, 0)
  break unless !"kill,quit".include? msg.chomp
end

client.close
```

Código 176: Cliente UDP

Rodando o servidor e o cliente:

```
$ ruby udpserver.rb
Servidor conectado na porta 12345, aguardando ...
Host 127.0.0.1 enviou um pacote UDP: oi
Host 127.0.0.1 enviou um pacote UDP: tudo bem?
Host 127.0.0.1 enviou um pacote UDP: kill
Kill recebido, fechando servidor.

$ ruby udpclient.rb
Digite sua mensagem (quit termina, kill finaliza servidor):
oi

Digite sua mensagem (quit termina, kill finaliza servidor):
```

[4] http://pt.wikipedia.org/wiki/Protocolo_UDP

```
tudo bem?

Digite sua mensagem (quit termina, kill finaliza servidor):
kill
```

> **Dica**
>
> No método `send` o argumento 0 é uma *flag* que pode usar uma combinação de constantes (utilizando um `or` binário das constantes presentes em `Socket::MSG`.

8.10 SMTP

O `SMTP` é um protocolo para o **envio** de emails, baseado em texto. Há uma classe `SMTP` pronta para o uso em Ruby:

```
require "net/smtp"
require "highline/import"

from = "eustaquiorangel@gmail.com"
pass = ask("digite sua senha:") { |q| q.echo = "*" }
to   = "eustaquiorangel@gmail.com"

msg =<<FIM
From: #{from}
Subject: Teste de SMTP no Ruby
Apenas um teste de envio de email no Ruby.
Falou!
FIM

smtp = Net::SMTP.new("smtp.gmail.com", 587)
smtp.enable_starttls

begin
  smtp.start("localhost", from, pass, :plain) do |smtp|
    puts "conexão aberta!"
    smtp.send_message(msg, from, to)
    puts "mensagem enviada!"
  end
rescue => exception
  puts "ERRO: #{exception}"
  puts exception.backtrace
end
```

Código 177: Enviando e-mails com SMTP

Rodando o programa:

```
$ ruby smtp.rb
digite sua senha:
********
conexão aberta!
mensagem enviada!
```

Dica

Na linha 3 requisitamos o módulo `highline`, que nos permite "mascarar"a digitação da senha na linha 6. Deve ser instalado como uma *gem*. Também precisamos de uma conta de email que permita conexão menos "insegura":

```
$ gem install highline
```

Para abrir uma conexão com o Gmail, como demonstrado, temos que indicar na conta do Gmail que é permitido o acesso por aplicações "inseguras", que não usam a autenticação em dois fatores do Gmail.

8.11 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:

```
require "net/pop"
require "highline/import"

user = "eustaquiorangel@gmail.com"
pass = ask("digite sua senha:") { |q| q.echo = "*" }

pop = Net::POP3.new("pop.gmail.com", 995)
pop.enable_ssl(OpenSSL::SSL::VERIFY_NONE)

begin
  pop.start(user, pass) do |pop|
    if pop.mails.empty?
      puts "Sem emails!"
      return
    end
    pop.each do |msg|
      puts msg.header
    end
  end
rescue => exception
  puts "ERRO: #{exception}"
end
```

Código 178: Lendo e-mails com POP3

Levar em consideração as mesmas configurações do Gmail indicadas anteriormente.

Rodando o programa:

```
$ ruby pop3.rb
digite sua senha:
********
Return-Path: <eustaquiorangel@gmail.com>
Received: from localhost ([186.222.196.152])
by mx.google.com with ESMTPS id x15sm1427881vcs.32.2016.07.06.14.14.13
(version=TLSv1/SSLv3 cipher=OTHER);
Wed, 06 Jul 14:14:17 -0700 (PDT)
Message-ID: <4e14d029.8f83dc0a.6a32.5cd7@mx.google.com>
Date: Wed, 06 Jul 14:14:17 -0700 (PDT)
From: eustaquiorangel@gmail.com
Subject: Teste de SMTP no Ruby
```

8.12 FTP

O FTP é um protocolo para a transmissão de arquivos. Vamos requisitar um arquivo em um servidor FTP:

```
require "net/ftp"

host = "ftp.gnu.org"
user = "anonymous"
pass = "eustaquiorangel@gmail.com"
file = "README"

begin
  Net::FTP.open(host) do |ftp|
    puts "Conexão FTP aberta."
    ftp.login(user, pass)

    puts "Requisitando arquivo ..."
    ftp.chdir("pub")
    ftp.get(file)
    puts "Download efetuado."

    puts File.read(file)
  end
rescue => exception
  puts "ERRO: #{exception}"
end
```

Código 179: Usando FTP

Rodando o programa:

```
$ ruby ftp.rb
Conexão FTP aberta.
Requisitando arquivo ...
Download efetuado.
Welcome to ftp.mozilla.org!
This is a distribution point for software and developer tools related to the
Mozilla project. For more information, see our home page:
...
```

Podemos também enviar arquivos utilizando o método `put(local, remoto)`.

8.13 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):

```
require "net/http"

host = Net::HTTP.new("eustaquiorangel.com", 80)
resposta = host.get("/")
return if resposta.message != "OK"
puts resposta.body.scan(/<h1>.*<\/h1>/)
```

Código 180: Lendo um site com HTTP

Rodando o programa:

```
$ ruby http1.rb
<h1><a href="/posts/fazendo_o_seu_projeto_brotar">Fazendo o seu projeto brotar</a></h1>
<h1>Artigos anteriores</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:

```
require "open-uri"

resposta = URI.open("http://eustaquiorangel.com")
puts resposta.read.scan(/<h1>.*<\/h1>/)
```

Código 181: Lendo um site com HTTP e OpenURI

Podemos melhorar o código usando um *parser* para selecionar os elementos. Lembrando que já utilizamos a `Nokokiri` para `XML`, podemos utilizar também para HTTP:

```
require "open-uri"
require "nokogiri"

doc = Nokogiri::HTML(URI.open("http://eustaquiorangel.com"))
puts doc.search("h1").map { |elemento| elemento.text }
```

Código 182: Lendo um site com HTTP e Nokogiri

Rodando o programa:

```
Eustáquio Rangel
Fazendo o seu projeto brotar
Artigos anteriores
```

> Reparem que agora veio mais um `H1`, que é justamente o único elemento desse tipo que tem um atributo `class`. A `Nokogiri` já trata esse tipo de coisas, que não temos na expressão regular que utilizamos anteriormente.

Aproveitando que estamos falando de `HTTP`, vamos ver como disparar um servidor web, o `WEBrick`, que deve ser instalado como uma *gem*:

```
$ gem install webrick
```

```
require "webrick"
include WEBrick

s = HTTPServer.new(Port: 2000, DocumentRoot: Dir.pwd)
trap("INT") { s.shutdown }
s.start
```

Código 183: Disparando um servidor web

Rodando o programa:

```
$ ruby webrick.rb
INFO  WEBrick 1.7.0
INFO  ruby 3.0.0 (2020-12-25) [x86_64-linux]
INFO  WEBrick::HTTPServer#start: pid=115709 port=2000
```

8.14 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 utilizar o HTTPS facilmente:

```
require "net/https"

begin
  site = Net::HTTP.new("postman-echo.com", 443)
  site.use_ssl = true

  site.start do |http|
    req = Net::HTTP::Get.new("/get?foo=bar")
    response = http.request(req)
    puts response.body
  end
rescue => exception
  puts "erro: #{e}"
end
```

Código 184: Utilizando HTTPS

8.15 SSH

O SSH é ao mesmo tempo um software 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:

```
$ 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:

```
require "net/ssh"
require "highline/import"

host = "eustaquiorangel.com"
user = "taq"
pass = ask("digite sua senha") { |q| q.echo = "*" }

begin
  Net::SSH.start(host, user, password: pass) do |session|
    puts "Sessão SSH aberta!"

    session.open_channel do |channel|
      puts "Canal aberto!"
      channel.on_data do |ch, data|
        puts "> #{data}"
      end
      puts "Executando comando ..."
      channel.exec "ls -lah"
    end
    session.loop
  end
rescue => exception
  puts "ERRO:#{exception}"
  puts exception.backtrace
end
```

Código 185: Utilizando SSH

Rodando o programa:

```
$ ruby ssh.rb
digite sua senha
*************
Sessão SSH aberta!
Canal aberto!
Executando comando
> total 103M
drwxr-xr-x 6 taq taq 4.0K Jun 17 19:10
...
```

8.16 Processos do sistema operacional

Podemos nos comunicar diretamente com o sistema operacional, executando comandos e recuperando as respostas.

8.16.1 Backticks

O jeito mais simples de fazer isso é com o uso de *backticks*:

```
> time = `date +%H:%M`
=> "16:26\n"

> puts time
16:26
```

O uso dos *backticks* fazem um *fork* do processo atual, executando o comando em um novo processo, criando uma **operação bloqueante**, esperando o comando terminar e o resultado é passado para o processo atual, podendo ser armazenado em uma variável. Se ocorrer algum erro no comando, esse erro é convertido em uma exceção:

```
> time = `xdate +%H:%M`
Errno::ENOENT: No such file or directory - xdate
        from (irb):3:in ``'
        from (irb):3
```

O uso de interpolação é permitido nas *backticks*:

```
> cmd = "date"
=> "date"

> time = `#{cmd} +%H:%M`
=> "16:30\n"

> puts time
16:30
```

8.16.2 System

Utilizar `system` é parecido com *backticks* mas com algumas diferenças:

- As exceções são "engolidas".
- O retorno é *booleano* ou nulo, com `true` indicando que o comando foi bem sucedido, `false` se não foi bem sucedido e `nil` indicando um erro na execução.

Vamos ver como funciona:

```
> time = system("date +%H:%M")
16:31
=> true
> puts time
true

> time = system("xdate +%H:%M")
=> nil
> puts time
=> nil
```

8.16.3 Exec

Utilizar `exec` substitui o processo atual pelo processo executando o comando. Então, se estivermos no `irb` e utilizarmos `exec`, vamos **sair do irb e ir para o processo com o comando sendo executado**, então muito cuidado com isso:

```
> time = exec("date +%H:%M")
21:16
$
```

No caso de ocorrer um erro é retornando `nil`:

```
> time = exec("xdate +%H:%M")
Errno::ENOENT: No such file or directory - xdate
        from (irb):5:in `exec'
        from (irb):5
```

8.16.4 IO.popen

Roda o comando em um processo novo e retorna os fluxos de entrada e saída conectados à um objeto `IO`:

```
> time = IO.popen("date +%H:%M").read
=> "16:32\n"
```

8.16.5 Open3

É o que dá controle mais granular para os fluxos de `IO` envolvidos. Vamos imaginar que temos o seguinte *shell script* para ler um nome digitado e mostrar o resultado na tela:

```
#!/bin/bash
echo "Digite seu nome: "
read nome
echo "Oi, $nome!"
```

Código 186: Lendo um nome com shell script

Executando o *script* e digitando algum nome:

```
$ nome.sh
Digite seu nome:
taq
Oi, taq!
```

Agora queremos interagir com o *script*, conseguindo enviar alguma coisa para o fluxo de entrada (`STDIN`), ler do fluxo de saída (`STDOUT`) e do fluxo de erros (`STDERR`). Podemos utilizar o módulo `Open3` para isso:

```
require "open3"

Open3.popen3("./nome.sh") do |stdin, stdout, stderr, thread|
  stdin.puts "taq"
  puts stdout.read
  puts "Rodei no processo #{thread.pid}"
  erro = stderr.read
  puts "Ocorreu o erro: #{erro}" if erro.size > 0
end
```

Código 187: Lendo um arquivo texto

Rodando o programa:

```
$ ruby nome.rb
Digite seu nome:
Oi, taq!
Rodei no processo 19210
```

No programa:

- Enviamos a `String` "taq"para o fluxo de entrada (`STDIN`), que estava esperando ser digitado algum nome.
- Lemos o fluxo de saída (`STDOUT`) com o resultado do programa.
- Mostramos o `pid` do processo que foi rodado.
- Verificamos o fluxo de erro (`STDERR`) se ocorreu algum erro, e se ocorreu, imprimimos ele na tela.

8.17 XML-RPC

`XML-RPC`[5] 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.

[5] 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.

Vamos instalar a *gem* `xmlrpc`:

```
$ gem install xmlrpc
```

Antes de mais nada, vamos criar um servidor que vai responder as nossas requisições, fazendo algumas operações matemáticas básicas, que serão **adição** e **divisão**:

```
require "webrick"
require "xmlrpc/server"

server = XMLRPC::Server.new(8081)

# somando números
server.add_handler("soma") do |n1, n2|
  { "resultado" => n1 + n2 }
end

# dividindo e retornando o resto
server.add_handler("divide") do |n1, n2|
  { "resultado" => n1 / n2, "resto" => n1 % n2 }
end

server.serve
```

Código 188: Servidor RPC

Rodando o programa:

```
$ ruby rpcserver.rb
INFO WEBrick 1.3.1
INFO ruby 1.9.2 (2010-08-18) [i686-linux]
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):

```
require "xmlrpc/client"

begin
  client = XMLRPC::Client.new("localhost", "/RPC2", 8081)
  resp = client.call("soma", 5, 3)
  puts "O resultado da soma é #{resp['resultado']}"

  resp = client.call("divide", 11, 4)
  puts "O resultado da divisao é #{resp['resultado']} e o resto é #{resp['resto']}"
rescue => exception
  puts "ERRO: #{exception}"
end
```

Código 189: Cliente RPC

```
$ ruby rpcclient.rb
O resultado da soma é 8
O resultado da divisao é 2 e o resto é 3
```

Vamos acessar agora o servidor em outras linguagens!

8.17.1 Python

```
# coding: utf-8
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8081/") as server:
    result = server.soma(5,3)
    print("O resultado da soma é:", result["resultado"])

    result = server.divide(11,4)
    print("O resultado da divisão é", result["resultado"], "e o resto é", result["resto"])
```

Código 190: Cliente RPC em Python

Rodando o programa:

```
$ python rpcclient.py
O resultado da soma é: 8
O resultado da divisão é 2 e o resto é 3
```

8.17.2 PHP

Um pouco mais de código para fazer em `PHP`:

Temos que ter instalado no `PHP` o suporte para `XML-RPC`, que no Ubuntu pode ser feito dessa forma:

```
$ sudo apt install php-xmlrpc
```

```php
<?php
// soma
$request = xmlrpc_encode_request("soma", [5, 3]);
$context = stream_context_create([
    "http" => [
        "method"  => "POST",
        "header"  => "Content-Type: text/xml",
        "content" => $request
]]);

$file     = file_get_contents("http://localhost:8081", false, $context);
$response = xmlrpc_decode($file);

if ($response && xmlrpc_is_fault($response)) {
    trigger_error("xmlrpc: ".$response["faultString"]." (".$response["faultCode"]." )");
} else {
    print "O resultado da soma é ".$response["resultado"]."\n";
}

// divisão
$request = xmlrpc_encode_request("divide", [11, 4]);
$context = stream_context_create([
    "http" => [
        "method"  => "POST",
        "header"  => "Content-Type: text/xml",
        "content" => $request
]]);

$file = file_get_contents("http://localhost:8081", false, $context);
$response = xmlrpc_decode($file);

if ($response && xmlrpc_is_fault($response)) {
    trigger_error("xmlrpc: ".$response["faultString"]." (".$response["faultCode"]." )");
} else {
    print "O resultado da divisão é ".$response["resultado"]." e o resto é ".$response["resto"]."\n";
}
?>
```

Código 191: Cliente RPC em PHP

Rodando o programa:

```
$ php rpcclient.php
O resultado da soma é 8
O resultado da divisão é 2 e o resto é 3
```

8.17.3 Java

Em `Java` vamos precisar do `Apache XML-RPC`[6], vamos pegar o último arquivo para rodar o seguinte código:

[6] https://ws.apache.org/xmlrpc/client.html

```
import java.net.URL;
import java.util.Vector;
import java.util.HashMap;
import org.apache.xmlrpc.common.*;
import org.apache.xmlrpc.client.*;

public class RPCClient {
  public static void main(String args[]) {
    try {
      Vector <Integer>params;
      XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
      config.setServerURL(new URL("http://localhost:8081/RPC2"));
      XmlRpcClient server = new XmlRpcClient();
      server.setConfig(config);

      params = new Vector<Integer>();
      params.addElement(new Integer(5));
      params.addElement(new Integer(3));

      HashMap result = (HashMap) server.execute("soma", params);
      int sum = ((Integer) result.get("resultado")).intValue();
      System.out.println("O resultado da soma é " + Integer.toString(sum));

      params = new Vector<Integer>();
      params.addElement(new Integer(11));
      params.addElement(new Integer(4));
      result = (HashMap) server.execute("divide", params);

      int divide = ((Integer) result.get("resultado")).intValue();
      int resto = ((Integer) result.get("resto")).intValue();
      System.out.println("O resultado da divisão é " + Integer.toString(sum) + " e o resto é: " + Integer.toS
    } catch(Exception error) {
      System.err.println("erro:" + error.getMessage());
    }
  }
}
```

Código 192: Cliente RPC em Java

Compilando e rodando o programa, especificando os arquivos `.jar` necessários:

```
$ javac -classpath ws-commons-util-1.0.2.jar:xmlrpc-client-3.1.3.jar:xmlrpc-common-3.1.3.jar: RPCClient.java

$ java -classpath ws-commons-util-1.0.2.jar:xmlrpc-client-3.1.3.jar:xmlrpc-common-3.1.3.jar: RPCClient
O resultado da soma é 8
O resultado da divisão é 8 e o resto é: 3
```

Capítulo 9

JRuby

Vamos instalar `JRuby` para dar uma olhada em como integrar Ruby com Java, usando a `RVM`:

```
$ rvm install jruby
$ rvm use jruby
$ jruby -v
$ jruby <versão>
```

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`:

```
$ export CLASSPATH=$CLASSPATH:$(locate -b '\jruby.jar'):.:
```

Se o comando `locate` não for encontrado/suportado, utilize `find`:

```
$ 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:

```
puts "digite seu nome:"
nome = gets.chomp
puts "oi, #{nome}!"
```

Código 193: Primeiro programa em JRuby

Vamos compilar o programa com o compilador do JRuby, o `jrubyc`:

```
$ jrubyc jruby.rb
```

E rodar o programa direto com Java!

```
$ java jruby
digite seu nome:
taq
oi, taq!
```

9.1 Utilizando classes do Java de dentro do Ruby

Vamos criar um programa chamado `gui.rb`:

```
# encoding: utf-8
require "java"

%w(JFrame JLabel JPanel JButton).each { |c| java_import("javax.swing.#{c}") }

class Alistener
  include java.awt.event.ActionListener
  def actionPerformed(event)
    puts "Botão clicado!"
  end
end
listener = Alistener.new

frame = JFrame.new
label = JLabel.new("Clique no botão!")
panel = JPanel.new

button = JButton.new("Clique em mim!")
button.addActionListener(listener)

panel.setLayout(java.awt.GridLayout.new(2,1))
panel.add(label)
panel.add(button)

frame.setTitle("Exemplo de JRuby")
frame.getContentPane().add(panel)
frame.pack
frame.defaultCloseOperation = JFrame::EXIT_ON_CLOSE
frame.setVisible(true)
```

Código 194: Usando interface gráfica em JRuby

Compilando e rodando o programa:

```
$ jrubyc gui.rb

$ java gui
```

Resulta em:

Pudemos ver que criamos a classe `Alistener` com a interface, no caso aqui com um comportamento de módulo, `java.awt.event.` 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`:

```
// Carro.java
public class Carro {
  private String marca, cor, modelo;
  private int tanque;

  public Carro(String marca, String cor, String modelo, int tanque) {
    this.marca  = marca;
    this.cor    = cor;
    this.modelo = modelo;
    this.tanque = tanque;
  }

  public String toString() {
    return "Marca: " + this.marca  + "\n" +
           "Cor: "   + this.cor    + "\n" +
           "Modelo:" + this.modelo + "\n" +
           "Tanque:" + this.tanque;
  }
}
```

Código 195: Classe Carro, em Java

e agora o arquivo `carro_java.rb`:

```
# carro.rb
require "java"
java_import("Carro")

carro = Carro.new("VW", "prata", "polo", 40)
puts carro

class Mach5 < Carro
  attr_reader :tanque_oxigenio

  def initialize(marca, cor, modelo, tanque, tanque_oxigenio)
    super(marca, cor, modelo, tanque)
    @tanque_oxigenio = tanque_oxigenio
  end

  def to_s
    "#{super}\nTanque oxigenio: #{@tanque_oxigenio}"
  end
end

puts "*" * 25
mach5 = Mach5.new("PopsRacer", "branco", "Mach5", 50, 10)
puts mach5
```

Código 196: Usando a classe Carro, em Java, dentro de Ruby

Compilando e rodando o programa:

```
$ javac Carro.java

$ jrubyc carro_java.rb

$ java carro_java

Marca: VW
Cor: prata
Modelo:polo
Tanque:40
```

```
************************
Marca: PopsRacer
Cor: branco
Modelo:Mach5
Tanque:50
Tanque oxigenio: 10
```

9.2 Usando classes do Ruby dentro do Java

Antes era um pouco mais complicado de fazer isso, mas parece que hoje está um pouco mais tranquilo. Primeiro vamos fazer o código em Ruby, que vai receber um valor do código em `Java` na variável `num`:

```
puts "Dobrando #{num} aqui no Ruby ..."
num * 2
```

Código 197: Código Ruby que vai ser chamado em Java

E agora o código `Java`, usando as *libs* do `JRuby`:

```
import org.jruby.*;
import org.jruby.embed.LocalVariableBehavior;
import org.jruby.embed.PathType;
import org.jruby.embed.ScriptingContainer;

public class Double {
   public static void main(String args[]) {
      try {
         ScriptingContainer container = new ScriptingContainer(LocalVariableBehavior.PERSISTENT);
         container.put("num", 2);

         String script = "doublejava.rb";
         System.out.println("Chamando o script " + script + " ...");
         long value = (Long) container.runScriptlet(PathType.CLASSPATH, script);
         System.out.println("Resultado: " + value);
      } catch(Exception e) {
         System.err.println("Erro: " + e.getMessage());
      }
   }
}
```

Código 198: Código Java que vai ser chamar o código Ruby

Compilando e rodando:

```
$ javac Double.java
$ java Double
Chamando o script doublejava.rb ...
Dobrando 2 aqui no Ruby ...
Resultado: 4
```

Capítulo 10

Banco de dados

Vamos utilizar uma interface uniforme para acesso aos mais diversos bancos de dados suportados em `Ruby` através da interface `Sequel`[1]. Para instalá-la, é só utilizar a *gem* `sequel`:

```
$ gem install sequel
```

Também vamos instalar a *gem* `sqlite3`, que nos dá suporte ao banco de dados auto-contido, sem servidor, com configuração zero e relacional (quanta coisa!) SQLite[2], que vai nos permitir testar rapidamente os recursos da `Sequel` sem precisar ficar configurando um banco de dados, já que o banco é criado em um arquivo simples no diretório corrente.

> **Atenção!**
>
> Pelo amor do ET de Varginha, não vão utilizar o SQLite para produção em alguma aplicação! É somente para pequenos bancos.

10.1 Abrindo a conexão

Vamos abrir e fechar a conexão com o banco:

```
require "sequel"
require "sqlite"

con = Sequel.sqlite(database: "alunos.sqlite3")
=> #<Sequel::SQLite::Database: {:adapter=>:sqlite, :database=>"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:

```
require "sequel"
require "sqlite3"

Sequel.sqlite(database: "alunos.sqlite3") do |con|
   p con
end
```

Desse modo sempre que a conexão for aberta, ela será automaticamente fechada no fim do bloco.

[1] http://sequel.rubyforge.org/
[2] https://sqlite.org/

Dica

Para trocar o banco de dados, podemos alterar apenas o método de conexão, ou seja, o `sqlite` ali após `Sequel`.

10.2 Consultas que não retornam dados

Vamos criar uma tabela nova para usamos no curso, chamada `alunos` e inserir alguns valores:

```
require 'sequel'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  con.run('drop table if exists alunos')

  sql = <<FIM
create table alunos (
id integer primary key autoincrement not null,
nome varchar(50) not null)
FIM
  con.run(sql)

  con[:alunos].insert(id: 1, nome: 'João')
  con[:alunos].insert(id: 2, nome: 'José')
  con[:alunos].insert(id: 3, nome: 'Antonio')
  con[:alunos].insert(id: 4, nome: 'Maria')
end
```

Código 199: Consultas que não retornam dados

Rodando o programa:

```
$ ruby db1.rb

$ sqlite3 alunos.sqlite3
SQLite
Enter ".help" for usage hints.
sqlite> select * from alunos;
1|João
2|José
3|Antonio
4|Maria
sqlite>
```

Apesar de ter criado a coluna `id` como auto-incremento, estou especificando os valores dela para efeitos didáticos.

10.3 Atualizando um registro

Aqui vamos utilizar o método `where` para selecionar o registro com o `id` que queremos atualizar, e o método `update` para fazer a atualização:

```
require 'sequel'
require 'sqlite3'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  puts con[:alunos].where(id: 4).update(nome: 'Mário')
end
```

Código 200: Atualizando um registro

Rodando o programa:

```
$ ruby db2.rb
1

$ sqlite3 alunos.sqlite3
SQLite
Enter ".help" for usage hints.
sqlite> select * from alunos where id = 4;
4|Mário
```

10.4 Apagando um registro

Vamos inserir um registro com o método `insert` e apagar com `delete`, após encontrar com `where`:

```
require 'sequel'
require 'sqlite3'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  con[:alunos].insert(id: 5, nome: 'Teste')
  puts con[:alunos].where(id: 5).delete
end
```

Código 201: Apagando um registro

Rodando o programa:

```
$ ruby db3.rb
1

$ sqlite3 alunos.sqlite3
SQLite
Enter ".help" for usage hints.
sqlite> select * from alunos where id = 5;
sqlite>
```

10.5 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":

```
require 'sequel'
require 'sqlite3'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  con[:alunos].each do |row|
    puts "id: #{row[:id]} nome: #{row[:nome]}"
  end
end
```

Código 202: Retornando dados

Rodando o programa:

```
$ ruby db4.rb
id: 1 nome: João
id: 2 nome: José
id: 3 nome: Antonio
id: 4 nome: Mário
```

Podemos recuperar todos as linhas de dados de uma vez usando `all`:

```
require 'sequel'
require 'sqlite3'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  rows = con[:alunos].all
  puts "#{rows.size} registros recuperados"
  rows.each do |row|
    puts "id: #{row[:id]} nome: #{row[:nome]}"
  end
end
```

Código 203: Retornando todos os registros

Rodando o programa:

```
$ ruby db5.rb
4 registros recuperados
id: 1 nome: João
id: 2 nome: José
id: 3 nome: Antonio
id: 4 nome: Mário
```

Ou se quisermos somente o primeiro registro:

```
require 'sequel'
require 'sqlite3'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  row = con[:alunos].first
  puts "id: #{row[:id]} nome: #{row[:nome]}"
end
```

Código 204: Retornando o primeiro registro

Rodando o programa:

```
$ ruby db6.rb
id: 1 nome: João
```

10.6 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:

```
require 'sequel'
require 'sqlite3'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  ds = con[:alunos].filter(id: :$i)
  ps = ds.prepare(:select, :select_by_id)

  (1..4).each do |id|
    print "procurando id #{id} ... "
    row = ps.call(i: id)
    puts "#{row.first[:nome]}"
  end
end
```

Código 205: Comandos preparados

Rodando o programa:

```
$ ruby db7.rb
procurando id 1 ... João
procurando id 2 ... José
procurando id 3 ... Antonio
procurando id 4 ... Mário
```

10.7 Metadados

Vamos dar uma examinada nos dados que recebemos de nossa consulta e na estrutura de uma tabela:

```
require 'sequel'
require 'sqlite3'

Sequel.sqlite(database: 'alunos.sqlite3') do |con|
  p con[:alunos].columns
  p con.schema(:alunos)
end
```

Código 206: Metadados da consulta

Rodando o programa:

```
$ ruby db8.rb

[:id, :nome]
[[:id, {:allow_null=>false, :default=>nil, :db_type=>"integer",
:primary_key=>true, :auto_increment=>true, :type=>:integer,
:ruby_default=>nil}], [:nome, {:allow_null=>false, :default=>nil,
:db_type=>"varchar(50)", :primary_key=>false, :type=>:string,
:ruby_default=>nil, :max_length=>50}]]
```

10.7.1 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`, já atendendo à uma pequena requisição do `ActiveRecord`, que pede uma coluna chamada `id` como chave primária, o que já temos. Só precisamos instalar a ‘gem‘ correspondente, que é a ‘active_record‘:

```
$ gem install activerecord
```

Dando uma olhada no programa:

```ruby
require 'active_record'

# estabelecendo a conexão
ActiveRecord::Base.establish_connection({
  adapter:  'sqlite3',
  database: 'alunos.sqlite3',
})

# criando o mapeamento da classe com a tabela
# (espera aí é só isso???)
class Aluno < ActiveRecord::Base
end

# pegando a coleção e usando o seu iterador
for aluno in Aluno.all
  puts "id: #{aluno.id} nome: #{aluno.nome}"
end

# atualizando o nome de um aluno
aluno = Aluno.find(3)
puts "encontrei #{aluno.nome}"
aluno.nome = 'Danilo'
aluno.save
```

Código 207: Utilizando ActiveRecord

Rodando o programa:

```
$ ruby db9.rb
id: 1 nome: João
id: 2 nome: José
id: 3 nome: Antonio
id: 4 nome: Maria
encontrei Antonio
```

Se rodarmos novamente, vamos verificar que o registro foi alterado, quando rodamos o programa anteriormente:

```
$ ruby db9.rb
id: 1 nome: João
id: 2 nome: José
id: 3 nome: Danilo
id: 4 nome: Maria
encontrei Danilo
```

Capítulo 11

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:

```
require "mkmf"

extension_name = "curso"
dir_config(extension_name)
create_makefile(extension_name)
```

Código 208: Criando o arquivo extconf.rb para o módulo em C

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:

```
#include <ruby.h>
#include <time.h>

VALUE modulo, classe;

void Init_curso(){
  modulo = rb_define_module("Curso");
  classe = rb_define_class_under(modulo, "Horario", rb_cObject);
}
```

Código 209: Iniciando o módulo em C

Opa! Já temos algumas coisas definidas ali! Agora temos que criar um `Makefile` [1] para compilarmos nossa extensão. O bom que ele é gerado automaticamente a partir do nosso arquivo `extconf.rb`:

[1] http://pt.wikibooks.org/wiki/Programar_em_C/Makefiles

```
$ ruby extconf.rb
creating Makefile
```

E agora vamos executar o `make` para ver o que acontece:

```
$ make
compiling curso.c
linking shared-object curso.so
```

Dando uma olhada no diretório, temos:

```
$ ls *.so
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:

```
$ irb
require_relative "curso"
=> true

> horario = Curso::Horario.new
=> #<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`:

```
#include <ruby.h>
#include <time.h>

VALUE modulo, classe;

VALUE t_init(VALUE self, VALUE valor){
  rb_iv_set(self, "@descricao", valor);
  return self;
}

void Init_curso(){
  modulo = rb_define_module("Curso");
  classe = rb_define_class_under(modulo, "Horario", rb_cObject);
  rb_define_method(classe,"initialize", t_init, 1);
}
```

Código 210: Construtor em C

Vamos testar, lembrando de rodar o `make` para compilar novamente o código:

```
require_relative "curso"
=> true

> horario = Curso::Horario.new
ArgumentError: wrong number of arguments(0 for 1)
from (irb):2:in ‘initialize’
from (irb):2:in ‘new’
from (irb):2
from /home/aluno/.rvm/rubies/ruby-1.9.2-p180/bin/irb:16:in ‘<main>’

> horario = Curso::Horario.new(:teste)
=> #<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:

```
horario.descricao
NoMethodError: undefined method ‘descricao’ for
#<Curso::Horario:0x8b9e5e4 @descricao=:teste>
from (irb):4
```

Vamos criar um método para acessá-la:

```
#include <ruby.h>
#include <time.h>

VALUE modulo, classe;

VALUE t_init(VALUE self, VALUE valor){
  rb_iv_set(self, "@descricao", valor);
  return self;
}

VALUE descricao(VALUE self){
  return rb_iv_get(self, "@descricao");
}

void Init_curso(){
  modulo = rb_define_module("Curso");
  classe = rb_define_class_under(modulo, "Horario", rb_cObject);
  rb_define_method(classe, "initialize", t_init, 1);
  rb_define_method(classe, "descricao", descricao, 0);
}
```

Código 211: Variáveis de instância em C

Rodando novamente:

```
require_relative "curso"
=> true

> horario = Curso::Horario.new(:teste)
=> #<Curso::Horario:0x8410d04 @descricao=:teste>

> horario.descricao
=> :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`.

```
#include <ruby.h>
#include <time.h>

VALUE modulo, classe;

VALUE t_init(VALUE self, VALUE valor){
  rb_iv_set(self, "@descricao", valor);
  return self;
}

VALUE descricao(VALUE self){
  return rb_iv_get(self, "@descricao");
}

struct tm *get_date_time() {
  time_t dt;
  struct tm *dc;
  time(&dt);
  dc = localtime(&dt);
  return dc;
}

VALUE data(VALUE self) {
  char str[30];
  struct tm *dc = get_date_time();
  sprintf(str, "%02d/%02d/%04d", dc->tm_mday, dc->tm_mon + 1, dc->tm_year + 1900);
  return rb_str_new2(str);
}

VALUE hora(VALUE self) {
  char str[15];
  struct tm *dc = get_date_time();
  sprintf(str, "%02d:%02d:%02d", dc->tm_hour, dc->tm_min, dc->tm_sec);
  return rb_str_new2(str);
}

void Init_curso() {
  modulo = rb_define_module("Curso");
  classe = rb_define_class_under(modulo, "Horario", rb_cObject);
  rb_define_method(classe, "initialize", t_init, 1);
  rb_define_method(classe, "descricao", descricao, 0);
  rb_define_method(classe, "data", data, 0);
  rb_define_method(classe, "hora", hora, 0);
}
```

Código 212: Métodos em C

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.

Rodando o programa:

```
require_relative "curso"
=> true

> horario = Curso::Horario.new(:teste)
=> #<Curso::Horario:0x896b6dc @descricao=:teste>
```

```
> horario.descricao
=> :teste

horario.data
=> "17/04/2021"

horario.hora
=> "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.html.

11.1 Utilizando bibliotecas externas

Vamos supor que precisamos fazer uma integração do nosso código Ruby com alguma `lib` externa, já pronta. Para isso temos que dar um jeito de acessar as funções dessa `lib` de dentro do nosso código Ruby. Aproveitando o código que vimos acima para recuperar a hora, vamos fazer uma pequena `lib`, chamada `libhora` que faz isso na função `hora`.

11.1.1 Escrevendo o código em C da lib

Para a `lib` vamos utilizar o seguinte código no arquivo `hora.c`:

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

struct tm *get_date_time() {
  time_t dt;
  struct tm *dc;
  time(&dt);
  dc = localtime(&dt);
  return dc;
}

char *hora() {
  char *str, cur[15];
  str = malloc(sizeof(char) * 15);
  struct tm *dc = get_date_time();
  sprintf(cur, "%02d:%02d:%02d", dc->tm_hour, dc->tm_min, dc->tm_sec);
  strcpy(str, cur);
  return str;
}
```

Código 213: Library externa em C

Compilando o programa para produzir o arquivo `hora.o`:

```
$ gcc -c -Wall -Werror -fpic hora.c
```

E agora convertendo para uma `lib` compartilhada, que vai produzir o arquivo `libhora.so`:

```
$ gcc -shared -o libhora.so hora.o
```

Para desencargo de consciência, vamos fazer código em C para utilizar essa `lib`, para o caso de acontecer algum problema e isolarmos direto em C para não achar que a causa é a integração com Ruby. Primeiro o arquivo *header* em `hora.h`:

```
#ifndef hora_h__
#define hora_h__

extern char* hora(void);

#endif  // hora_h__
```

Código 214: Header da library externa em C

E agora o programa de teste em `main.c`:

```
#include <stdio.h>
#include "hora.h"

int main(void)
{
    puts("Teste da lib compartilhada:");
    puts(hora());
    return 0;
}
```

Código 215: Arquivo para conferência da library em C

Compilando o programa de testes:

```
$ gcc -o main main.c -lhora -L$(pwd)
```

Para rodar o programa para testar, temos que indicar onde encontrar a `lib` compartilhada (que foi feito na compilação ali acima utilizando `-L` seguido de `pwd`):

```
$ LD_LIBRARY_PATH=$LB_LIBRARY_PATH:$(pwd) ./main
Teste da lib compartilhada:
20:05:54
```

Pronto, agora podemos testar no código Ruby.

11.2 Utilizando a lib compartilhada

Agora vamos utilizar essa `lib` dentro do nosso código Ruby. Para isso, vamos utilizar o módulo `fiddle`, com o seguinte programa:

```
require "fiddle"

# carrega a lib compartilhada
libhora = Fiddle.dlopen("./libhora.so")

# pega uma referência para a função
hora = Fiddle::Function.new(libhora["hora"], [], Fiddle::TYPE_VOIDP)

# chama a função
puts hora.call
```

Código 216: Utilizando libraries externas em C

Rodando o programa vemos que tudo correu bem:

```
$ ruby fiddle.rb
20:10:27
```

Temos que adequar as requisições para as referências e chamadas de funções para o número e tipo correto de valores que vamos enviar e receber. Para mais informações de como fazer isso na documentação do Fiddle.

Capítulo 12

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 assigments*.

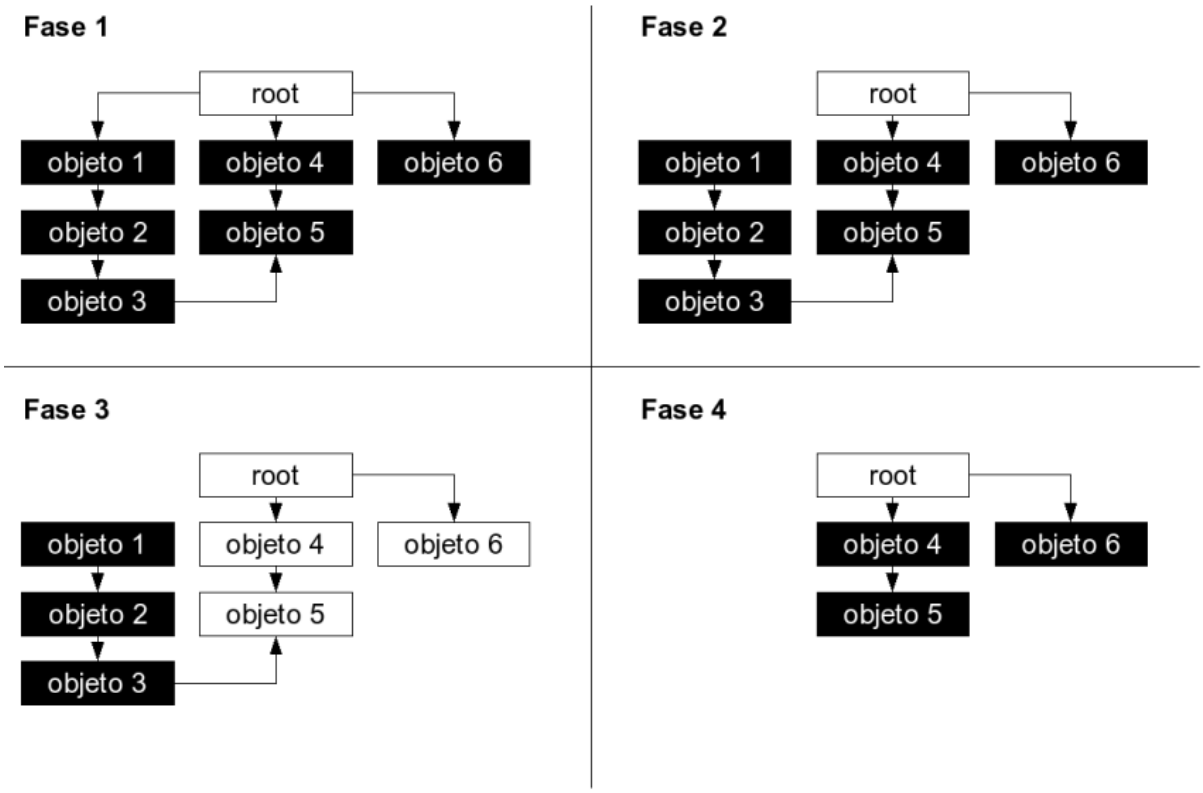
Em Ruby, o *garbage collector* é *basicamente* do tipo *mark-and-sweep*, que atua em fases separadas onde marca os objetos que não são mais necessários e depois os limpa. Vamos ver fazendo um teste prático de criar alguns objetos, invalidar algum, chamar o *garbage collector* e verificar os objetos novamente:

```
class Teste
end

t1 = Teste.new
t2 = Teste.new
t3 = Teste.new

count = ObjectSpace.each_object(Teste) do |object|
  puts object
end
puts "#{count} objetos encontrados."

t2 = nil
GC.start

count = ObjectSpace.each_object(Teste) do |object|
  puts object
end
puts "#{count} objetos encontrados."
```

Código 217: Garbage collector

Rodando o programa:

```
$ ruby gc1.rb

#<Teste:0x850d1a8>
#<Teste:0x850d1bc>
#<Teste:0x850d1d0>
3 objetos encontrados.
#<Teste:0x850d1a8>
#<Teste:0x850d1d0>
2 objetos encontrados.
```



- 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.

Os gatilhos para disparar o *garbage collector* são definidos na VM, como por exemplo quando a alocação de um novo objeto excede um determinado limite/*threshold*, onde é iniciado um novo ciclo de coleta. Como vimos no exemplo acima, podemos disparar essa coleta/ciclo através de `GC.start`, mas é **muito boa idéia** deixar isso a cargo da VM.

Vamos dar uma olhada em algumas estatísticas do *garbage collector* utilizando `GC.stat` antes e depois de executarmos o `GC.start`:

```ruby
class Teste
end

t1 = Teste.new
t2 = Teste.new
t3 = Teste.new

count = ObjectSpace.each_object(Teste) do |object|
  puts object
end
puts "#{count} objetos encontrados."

t2 = nil

puts "Antes:"
p GC.stat

GC.start

puts "Depois:"
p GC.stat

count = ObjectSpace.each_object(Teste) do |object|
  puts object
end
puts "#{count} objetos encontrados."
```

Código 218: Estatísticas do garbage collector

Rodando o programa, vamos ver algo como:

```
$ ruby gc2.rb
#<Teste:0x000055e987b64628>
#<Teste:0x000055e987b64678>
#<Teste:0x000055e987b646a0>
3 objetos encontrados.

Antes:

{:count=>9, :heap_allocated_pages=>49, :heap_sorted_length=>49,
:heap_allocatable_pages=>0, :heap_available_slots=>20021,
:heap_live_slots=>18102, :heap_free_slots=>1919, :heap_final_slots=>0,
:heap_marked_slots=>15479, :heap_eden_pages=>49, :heap_tomb_pages=>0,
:total_allocated_pages=>49, :total_freed_pages=>0,
:total_allocated_objects=>55021, :total_freed_objects=>36919,
:malloc_increase_bytes=>97776, :malloc_increase_bytes_limit=>16777216,
:minor_gc_count=>8, :major_gc_count=>1, :compact_count=>0,
:read_barrier_faults=>0, :total_moved_objects=>0,
:remembered_wb_unprotected_objects=>213,
:remembered_wb_unprotected_objects_limit=>290, :old_objects=>15185,
:old_objects_limit=>23716, :oldmalloc_increase_bytes=>411912,
:oldmalloc_increase_bytes_limit=>16777216}

Depois:

{:count=>10, :heap_allocated_pages=>50, :heap_sorted_length=>65,
:heap_allocatable_pages=>15, :heap_available_slots=>20429,
:heap_live_slots=>16064, :heap_free_slots=>4365, :heap_final_slots=>0,
:heap_marked_slots=>16062, :heap_eden_pages=>50, :heap_tomb_pages=>0,
:total_allocated_pages=>50, :total_freed_pages=>0,
:total_allocated_objects=>55087, :total_freed_objects=>39023,
```

```
:malloc_increase_bytes=>2616, :malloc_increase_bytes_limit=>16777216,
:minor_gc_count=>8, :major_gc_count=>2, :compact_count=>0,
:read_barrier_faults=>0, :total_moved_objects=>0,
:remembered_wb_unprotected_objects=>205,
:remembered_wb_unprotected_objects_limit=>410, :old_objects=>15828,
:old_objects_limit=>31656, :oldmalloc_increase_bytes=>321328,
:oldmalloc_increase_bytes_limit=>16777216}
#<Teste:0x000055e987b64628>
#<Teste:0x000055e987b646a0>
2 objetos encontrados.
```

Podemos ver ali várias mudanças, entre elas, a contagem de vezes que o *garbage collector* foi executado em `count`.

Uma curiosidade é que a partir da versão `2.2`, as referências do tipo `Symbol`, que antes não eram coletadas, agora são. Vamos fazer um teste rápido:

```
before =  Symbol.all_symbols.size

100_000.times do |i|
  "sym#{i}".to_sym
end

puts Symbol.all_symbols.size

GC.start

puts Symbol.all_symbols.size
```

Código 219: Coletando símbolos no garbage collector

Rodando o programa, vamos ver algo como:

```
$ ruby sym_gc.rb
4712
3196
```

Importante notar que tem alguns símbolos considerados pela VM como "imortais", especialmente os que ela utiliza de forma interna.

12.1 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:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char *str;
  str = malloc(sizeof(char) * 15);
  strcpy(str, "hello world");
  printf("%s\n", str);
  free(str);
  return 0;
}
```

Código 220: Garbage collector

Vamos compilá-lo (você tem o `GCC` aí, não tem?) e executá-lo:

```
$ gcc -o null null.c
$ ./null
hello world
```

Até aqui tudo bem. Mas agora comentem a linha 7, onde é executada `malloc`:

```
$ gcc -o null null.c
$ ./null
hello world
*** Error in `./null': free(): invalid pointer: 0xb7758000 ***
======= Backtrace: =========
...
```

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`:

```
$ gcc -o null null.c
$ ./null
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 ...

12.2 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 `Valgrind`[1], 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:

```
$ valgrind --tool=memcheck --leak-check=yes -q ./null
==8119== Use of uninitialised value of size 4
==8119==    at 0x8048429: main (in /home/taq/code/ruby/conhecendo-ruby/null)
==8119==
...
```

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:

```
$ gcc -o null null.c
$ valgrind --tool=memcheck --leak-check=yes -q ./null
==8793== Use of uninitialised value of size 4
==8793==    at 0x8048459: main (in /home/taq/code/ruby/conhecendo-ruby/null)
==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`:

```
$ gcc -o null null.c
$ valgrind --tool=memcheck --leak-check=yes -q ./null
hello world
```

Agora temos certeza de que está tudo ok! O `Valgrind` é uma ferramenta muito poderosa que quebra altos galhos.

[1] http://valgrind.org

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:

```
$ gcc -g -o null null.c
$ valgrind --tool=memcheck --leak-check=yes -q ./null
==9029== Use of uninitialised value of size 4
==9029==    at 0x8048459: main (null.c:8)
==9029==
```

Reparem que agora ele já dedurou que o problema está na linha 8 (`null.c:8`), onde está sendo copiado um valor para uma variável não alocada.

12.2.1 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` para cima) são alocadas com `malloc`, diretamente no *heap*, que são os casos das chamadas "Strings de heap", mas existem também as "Strings compartilhadas", que são `Strings` que apontam para outras, ou seja, quando utilizamos algo como `str2 = str1`, vão apontar para o mesmo local e um outro tipo de `Strings`, as consideradas "Strings embutidas"(*"embedded"*), que são descritas a seguir.

Os objetos em Ruby são basicamente criados todos na área de *heap* da memória, onde ocorre a alocação de memória com `malloc` e posterior coleta pelo garbage collector dos objetos que não estão sendo mais utilizados. Se você tem alguma dificuldade de decorar em que área da memória utilizamos `malloc`, decore a expressão "hippie maloqueiro", visualize a imagem e nunca mais você vai esquecer disso.

Essas `Strings` tem limites de até 11 caracteres em máquinas 32 *bits* e 23 caracteres em máquinas 64 *bits*, 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`, consequentemente, aumentando a velocidade. O nosso programa acima seria algo como:

```c
#include <stdio.h>

int main() {
  char str[15] = "hello world";
  printf("%s\n", str);
  return 0;
}
```

Código 221: Strings com tamanho definido

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:

```c
#include <stdio.h>
#include <limits.h>

int main() {
   int size = ((int) ((sizeof(void *) * 3) / sizeof(char) - 1));
   printf("%d bits: %d bytes de comprimento\n", __WORDSIZE, size);
}
```

Código 222: Limites de String

O resultado vai ser algo como, em computadores com 64 bits:

```
64 bits: 23 bytes de comprimento
```

Se quisermos simular um ambiente de 32 bits, primeiro temos que instalar os recursos necessários para isso:

```
$ sudo apt install gcc-multilib
```

E compilar e rodar o programa dessa forma:

```
$ gcc -m32 -o gc3 gc3.c
$ ./gc3
32 bits: 11 bytes de comprimento
```

A sequência que Ruby vai fazer para definir uma `String` é basicamente:

- Se for uma cópia de outra `String`, vai ser criada uma `String` compartilhada, onde só vamos precisar de uma área de memória para armazenar os dados dela.
- Se a `String` cabe dentro dos valores que vimos acima, é criada uma `String` embutida/embedded, que é mais rápido do que alocar e desalocar a memória, apenas copiando para a estrutura que dá suporte à esse tipo de `String` (vista logo ali abaixo).
- Se não for nenhum desses casos acima, é criada uma `String` de *heap*, utilizando `malloc` para alocar o espaço (olhem aí o hippie maloqueiro de novo!).

Vamos fazer um *benchmark* para testar isso:

```ruby
require 'benchmark'

Benchmark.bm do |bm|
  bm.report('alocando strings com menos de 23 caracteres') do
    1_000_000.times { s = '*' * 10 }
  end

  bm.report('alocando strings com 23 caracteres') do
    1_000_000.times { s = '*' * 23 }
  end

  bm.report('alocando strings com 24 caracteres') do
    1_000_000.times { s = '*' * 24 }
  end

  bm.report('alocando strings com mais de 24 caracteres') do
    1_000_000.times { s = '*' * 100 }
  end
end
```

Código 223: Benchmark com Strings de tamanhos diversos

Rodando o programa, podemos ver como existem similaridades entre as `Strings` embutidas e as alocadas, independente dos tamanhos das categorias:

```
$ ruby str_bench.rb
                                               user     system      total        real
alocando strings com menos de 23 caracteres 0.149636   0.000000   0.149636 (  0.149654)
alocando strings com 23 caracteres          0.147452   0.000000   0.147452 (  0.147464)
alocando strings com 24 caracteres          0.213678   0.000000   0.213678 (  0.213690)
alocando strings com mais de 24 caracteres  0.218274   0.000000   0.218274 (  0.218280)
```

Como curiosidade, essa é a estrutura que cuida de `Strings` no código de Ruby, `RString`, que é um dos tipos definidos como `RVALUE` na VM:

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,

```
struct RString {

  struct RBasic basic;

  union {
    struct {
      long len;
      char *ptr;
      union {
        long capa;
        VALUE shared;
      } aux;
    } heap;

    char ary[RSTRING_EMBED_LEN_MAX + 1];
  } as;
};
```

Código 224: Limites

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`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

union data {
  int id;
  char name[20];
};

int main() {
  union data d;
  d.id = 1;
  strcpy(d.name, "taq");
  printf("%d %s\n", d.id, d.name);
  return 0;
}
```

Código 225: Utilizando unions em C

Rodando o programa, temos algo como isso:

```
$ ./union
7430516 taq
```

Agora, se utilizarmos cada membro da `union` **de cada vez**, temos o comportamento esperado:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

union data {
  int id;
  char name[20];
};

int main() {
  union data d;

  d.id = 1;
  printf("%d\n", d.id);

  strcpy(d.name, "taq");
  printf("%s\n", d.name);

  return 0;
}
```

Código 226: Utilizando unions corretamente em C

Rodando o programa:

```
$ ./union2
1
taq
```

Capítulo 13

Testes

Se você está aprendendo Ruby para utilizar depois 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.

Os testes não são só uma metodologia de decoreba. São uma cultura, uma prática, algo que deve ser transformado em um hábito. Deve ser algo que se incorpore de maneira transparente no seu dia-a-dia de desenvolvimento de código, o que com certeza vai aumentar muito a qualidade do seu código, a sua produtividade com o desenvolvimento do mesmo e a garantia de estar fazendo um investimento bem sólido para o futuro do seu projeto.

No filme "Shine - Brilhante", baseado na vida do pianista David Helfgott, interpretado de forma brilhante (o que refletiu no título em Português) por Geoffrey Rush, na hora de ensinar como tocar uma obra no piano, o professor diz que primeiro devemos decorar cada nota, e quando estivemos bons o suficiente, devemos esquecer as notas e aí sim tocar com a emoção. Desenvolver código orientado à testes é mais ou menos esse esquema: aprendemos *bem* o jeito de se fazer e depois ficamos tão acostumados que absorvemos sem esforço depois. Filme altamente recomendado, mostrando o protagonista que é um gênio na música e todos os perrengues que ele passou, além de música de ótima qualidade. A obra de Rachmaninoff é de uma beleza singular.

Para isso, é importante escolher uma ferramenta de testes que não seja difícil, burocrática ou que fique mudando de sintaxe em toda versão. Fazer testes deve ser prazeroso e agregar valor para o desenvolvedor que usa a ferramenta. Se uma ferramenta de testes é muito complicada de utilizar, acaba gerando uma resistência em fazer testes e isso é muito ruim. Por sorte em Ruby temos várias ferramentas bem eficientes para isso.

Vamos usar de exemplo uma calculadora que só tem soma e subtração, então vamos fazer uma classe para ela, no arquivo `calc.rb`:

```
class Calculadora
  def soma(a, b)
    a + b
  end

  def subtrai(a, b)
    a - b
  end

  def media(colecao)
    val = colecao.valores
    val.reduce(:+) / val.size.to_f
  end
end
```

Código 227: Calculadora para testes

E agora o nosso teste propriamente dito, no arquivo `calc_test.rb`:

```ruby
require 'test/unit'
require_relative 'calc'

class TesteCalculadora < Test::Unit::TestCase
  def setup
    @calculadora = Calculadora.new
  end

  def test_adicao
    assert_equal(2, @calculadora.soma(1, 1), '1 + 1 = 2')
  end

  def test_subtracao
    assert_equal(0, @calculadora.subtrai(1, 1), '1 - 1 = 0')
  end

  def teardown
    @calculadora = nil
  end
end
```

Código 228: Testes para a calculadora

> Deixei um método da calculadora, `media`, sem testes agora de propósito, para criarmos um teste mais tarde, mas **sempre testem todos os métodos do objeto**.

Rodando os testes:

```
$ ruby calc_test.rb
Loaded suite calc_test
Started
..
Finished in 0.000407485 seconds.
------------------------------------------------------------------------------
2 tests, 2 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
------------------------------------------------------------------------------
4908.16 tests/s, 4908.16 assertions/s
```

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 da classe `Calculadora` ficar meio maluco, por exemplo, retornando o resultado mais 1, e rode os testes novamente:

```
$ ruby calc_test.rb
Loaded suite calc_test
Started
.F
===============================================================
     11:   end
     12:
     13:   def test_subtracao
```

```
  => 14:      assert_equal(0, @calculadora.subtrai(1, 1), '1 - 1 = 0')
     15:    end
     16:
     17:    def teardown
calc_test.rb:14:in `test_subtracao'
1 - 1 = 0
<0> expected but was
<2>
Failure: test_subtracao(TesteCalculadora)
================================================================================

Finished in 0.004781519 seconds.
--------------------------------------------------------------------------------
2 tests, 2 assertions, 1 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
50% passed
--------------------------------------------------------------------------------
418.28 tests/s, 418.28 assertions/s
```

No resultado é demonstrado exatamente o que falhou: na linha 14, o resultado esperado era 0, mas foi diferente, como indicado em `<0> expected but as <2>`, ou seja, era esperado 0 mas foi retornado 2.

No resumo, foram contabilizados 2 testes, 2 asserções, 1 falha, nenhum erro, testes pendentes, omissões ou notificações, sendo que:

- Testes são a quantidade de métodos de teste (que começam com `test_`) encontrados no arquivo.
- A quantidade de asserções (métodos que testam algo, que começam com `assert_`) encontradas no arquivo.
- A quantidade falhas encontradas, 1 no caso acima, onde uma asserção falhou.
- A quantidade de erros encontrados. Os errors são contabilizados se por acaso alguma coisa quebrou no código testado ou no código do teste. Experimentem trocar `a - b` no método de subtração para `a - b - x` para verem um erro disparar e ser contabilizado.
- A quantidade de testes pendentes. Podemos marcar algum teste dessa forma se por acaso ainda não temos o código pronto ou ele precisa de determinada condição. Isso é importante pois quando fazemos os testes já temos em mente como o código deve se comportar e devemos expressar isso nos testes até escrever o código para satisfazer o teste. As pendências são também como um marcador TODO para ficar nos enchendo o saco até finalmente escrever o código para remover o marcador, mas é **muito importante** já deixar isso parametrizado. Um exemplo de teste pendente pode ser visto abaixo.
- A quantidade de testes que foram omitidos. Isso pode ser utilizado para omitir testes, mas **deixar isso bem claro na suíte de testes**, por causa de alguma coisa que ainda está errada ou por um determinado comportamento. Um exemplo de teste omitido pode ser visto abaixo.
- A quantidade de notificações. As notificações podem ser utilizadas como mensagens de debug durante os testes, para acompanharmos determinadas coisas que podem acontecer durante os testes. Um exemplo de notificações pode ser visto abaixo.

Exemplo de teste pendente:

```
require 'test/unit'
require_relative 'calc'

class TesteCalculadora < Test::Unit::TestCase
  def setup
    @calculadora = Calculadora.new
  end

  def test_adicao
    assert_equal(2, @calculadora.soma(1, 1), '1 + 1 = 2')
  end

  def test_subtracao
    assert_equal(0, @calculadora.subtrai(1, 1), '1 - 1 = 0')
  end

  def test_raiz_quadrada
    pend('Ainda não fizemos esse método')
    assert_equal(2, @calculadora.raiz_quadrada(4), '2 é raiz de 4')
  end

  def teardown
    @calculadora = nil
  end
end
```

Código 229: Teste pendente

Exemplo de testes omitidos:

```
require 'test/unit'
require_relative 'calc'

class TesteCalculadora < Test::Unit::TestCase
  def setup
    @calculadora = Calculadora.new
  end

  def test_adicao
    assert_equal(2, @calculadora.soma(1, 1), '1 + 1 = 2')
  end

  def test_subtracao
    assert_equal(0, @calculadora.subtrai(1, 1), '1 - 1 = 0')
  end

  def test_raiz_quadrada
    omit('Fugi da escola, não sei fazer isso')
    assert_true(1 == 1)
  end

  def test_linux
    omit_unless('Só funciona no Linux', RUBY_PLATFORM.match?(/linux/i))
    assert_true(Dir.exist?('/tmp'))
  end

  def test_tarde
    omit_if('Só roda à tarde', Time.now.hour >= 12)
    assert_true(Time.hour.hour >= 12)
  end

  def teardown
    @calculadora = nil
  end
end
```

Código 230: Testes omitidos

Exemplo de testes com notificações:

```
require 'test/unit'
require_relative 'calc'

class TesteCalculadora < Test::Unit::TestCase
  def setup
    @calculadora = Calculadora.new
  end

  def test_adicao
    assert_equal(2, @calculadora.soma(1, 1), '1 + 1 = 2')
  end

  def test_subtracao
    notify('Começando o teste de subtração')
    assert_equal(0, @calculadora.subtrai(1, 1), '1 - 1 = 0')
    notify('Teste de subtração terminado')
  end

  def teardown
    @calculadora = nil
  end
end
```

Código 231: Testes com notificações

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
- assert_true

Vamos incluir algumas outras:

```ruby
require "test/unit"
require_relative "calc"

class TesteCalculadora < Test::Unit::TestCase
  def setup
    @calculadora = Calculadora.new
  end

  def test_objeto
    assert_kind_of Calculadora, @calculadora
    assert_match /^\d$/, @calculadora.soma(1, 1).to_s
    assert_respond_to @calculadora, :soma
    assert_same @calculadora, @calculadora
  end

  def test_objetos
    assert_operator @calculadora.soma(1, 1), :>, @calculadora.soma(1, 0)
  end

  def test_adicao
    assert_equal 2, @calculadora.soma(1, 1), "1 + 1 = 2"
  end

  def test_subtracao
    assert_equal 0, @calculadora.subtrai(1, 1), "1 - 1 = 0"
  end

  def teardown
    @calculadora = nil
  end
end
```

Código 232: Testes com mais asserções

Rodando os novos testes:

```
%$ ruby calc_test2.rb
Loaded suite calc_test2
Started
....
Finished in 0.000745739 seconds.
------------------------------------------------------------------------------
4 tests, 7 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
------------------------------------------------------------------------------
```

13.1 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`:

```ruby
require 'minitest/autorun'
require_relative 'calc'

class TesteCalculadora < Minitest::Test
  def setup
    @calculadora = Calculadora.new
  end

  def teardown
    @calculadora = nil
  end

  def test_objeto
    assert_kind_of Calculadora, @calculadora
    assert_match /^\d$/, @calculadora.soma(1, 1).to_s
    assert_respond_to @calculadora, :soma
    assert_same @calculadora, @calculadora
  end

  def test_objetos
    assert_operator @calculadora.soma(1, 1), :>, @calculadora.soma(1, 0)
  end

  def test_adicao
    assert_equal 2, @calculadora.soma(1, 1), "1 + 1 = 2"
  end

  def test_subtracao
    assert_equal 0, @calculadora.subtrai(1, 1), "1 - 1 = 0"
  end
end
```

Código 233: Utilizando Minitest

Mas que? Aparentemente só mudou no código de onde herdávamos de `Test::Unit::TestCase` e agora é `Minitest::Test`?

13.1.1 Randomizando os testes

Qual a vantagem? Antes de mais nada, vamos rodar o teste para ver o resultado:

```
Run options: --seed 21510

# Running:

....

Finished in 0.001438s, 2781.7882 runs/s, 5563.5764 assertions/s.

4 runs, 8 assertions, 0 failures, 0 errors, 0 skips
```

Reparem em `-seed 21510`. 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 previnir 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:

```
$ ruby minitest1.rb --seed 21510
```

13.1.2 Testando com specs

Também podemos testar utilizando *specs*, no estilo do `RSpec`, reescrevendo o código dessa maneira:

```
require 'minitest/autorun'
require_relative 'calc'

describe Calculadora do
  before do
    @calculadora = Calculadora.new
  end

  after do
    @calculadora = nil
  end

  describe 'objeto' do
    it 'deve ser do tipo de Calculadora' do
      expect(@calculadora).must_be_kind_of Calculadora
    end

    it 'deve ter um método para somar' do
      expect(@calculadora).must_respond_to :soma
    end

    it 'deve ter um método para subtrair' do
      expect(@calculadora).must_respond_to :subtrai
    end
  end

  describe 'soma' do
    it 'deve ser igual a 2' do
      expect(@calculadora.soma(1, 1)).must_equal 2
    end
  end

  describe 'subtração' do
    it 'deve ser igual a 0' do
      expect(@calculadora.subtrai(1, 1)).must_equal 0
    end
  end
end
```

Código 234: Testes da calculadora, com specs

Rodando o programa:

```
$ ruby calc_spec.rb
Run options: --seed 50323

# Running:

.....

Finished in 0.001507s, 3317.3327 runs/s, 3317.3327 assertions/s.

5 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Agora já mudou bastante o código dos testes, mais organizados e contextualizados, mas não a saída, que continua a mesma. Podemos usar alguns atalhos como `let`, ao invés do método `before`, que é um método **lazy** e só executa o bloco quando é invocado:

```
require 'minitest/autorun'
require_relative 'calc'

describe "Calculadora" do
   let(:calculadora) { Calculadora.new }
   ...
```

Podemos pular algum teste (lembram dos omitidos acima?), utilizando `skip`:

```
it "deve ter um método para multiplicar" do
   skip "ainda não aprendi como multiplicar"
   calculadora.must_respond_to :multiplicar
end
```

13.1.3 Benchmarks

O `Minitest` já vem com recursos de *benchmarks*:

```
require 'minitest/autorun'
require 'minitest/benchmark'
require_relative 'calc'

describe 'Calculadora Benchmark' do
  before do
    @calculadora = Calculadora.new
  end

  bench_performance_constant 'primeiro algoritmo', 0.001 do |n|
    100.times do |v|
      @calculadora.soma(n, v)
    end
  end

  bench_performance_constant 'segundo algoritmo', 0.001 do |n|
    100.times do |v|
      @calculadora.soma(v, n)
    end
  end
end
```

Código 235: Testes com benchmark

Rodando o programa:

```
$ ruby calc_bench_spec.rb
Run options: --seed 14916

# Running:

bench_primeiro_algoritmo 0.000024 0.000019 0.000017 0.000013 0.000024
bench_segundo_algoritmo  0.000022 0.000012 0.000013 0.000011 0.000019

Finished in 0.042308s, 47.2726 runs/s, 47.2726 assertions/s.

2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

13.2 Mocks

Temos um sistema básico e fácil para utilizar `mocks` [1], onde podemos simular o comportamento de um objeto complexo, ainda não acessível ou construído ou impossível de ser incorporado no teste, e verificar se os métodos dele foram acionados. Um `mock` é recomendado se:

- Gera resultados não deterministicos (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, que tem vários recursos interessantes, mas com o `Minitest` grande parte do que precisamos já está pronto.

Agora vamos utilizar o método chamado `media`, que vai receber e calcular a média de uma coleção e 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`:

[1] http://pt.wikipedia.org/wiki/Mock_Object

```
require 'minitest/autorun'
require_relative 'calc'

describe Calculadora do
  before do
    @calculadora = Calculadora.new
  end

  after do
    @calculadora = nil
  end

  describe 'média' do
    it 'deve ser igual a 2' do
      colecao = MiniTest::Mock.new
      colecao.expect :valores, [1, 2, 3]
      @calculadora.media(colecao)
      colecao.verify
    end
  end
end
```

Código 236: Mocks

"Falsificamos"um objeto, com um método chamado `valores`, que retorna um `Array` de 3 `Fixnum's`: `[1,2,3]`. A instrução ali é algo como "ei, quando o método `valores` for acionado em `colecao`, retorne aquele `Array` que indicamos e **verifique** que ele foi chamado". A verificação é feita pelo método `verify`.

13.3 Stubs

Também podemos ter `stubs`[2], 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:

```
require 'minitest/autorun'
require_relative 'calc'

describe Calculadora do
  before do
    @calculadora = Calculadora.new
  end

  after do
    @calculadora = nil
  end

  describe 'soma maluca' do
    it 'deve ser igual a 3' do
      @calculadora.stub :soma, 3 do
        expect(@calculadora.soma(1, 1)).must_equal 3
      end
    end
  end
end
```

Código 237: Stubs

Vejam que aqui eu só substitui temporariamente o método, sem verificar que ele estava sendo chamado de alguma forma, como fizemos com o `mock`. No `mock` é testado um comportamento enquanto que no `stub` está sendo definido esse comportamento.

[2] http://pt.wikipedia.org/wiki/Stub

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`.

13.4 Expectations

Vamos ver algumas das `expectations` [3] 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:

```
10.must_be :<, 20
```

- **must_be_empty** - Deve ser vazio:

```
[].must_be_empty
```

- **must_be_instance_of** - Deve ser uma instância de uma classe:

```
"oi".must_be_instance_of String
```

- **must_be_kind_of** - Deve ser de um determinado tipo:

```
1.must_be_kind_of Numeric
```

- **must_be_nil** - Deve ser nulo:

```
a = nil
a.must_be_nil
```

- **must_be_same_as** - Deve ser o mesmo objeto:

```
a = "oi"
b = a
a.must_be_same_as b
```

- **must_be_silent** - O bloco não pode mandar nada para `stdout` ou `stderr`:

```
-> {}.must_be_silent
=> true
-> { puts "oi" }.must_be_silent
1) Failure:
test_0002_should be silent(Test) [minitest.rb:10]:
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 o valor absoluto de `exp-act` (`delta>(exp-act).abs`):

```
1.01.must_be_within_delta 1.02, 0.1
=> true
1.01.must_be_within_delta 1.02, 0.1
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`).

[3] http://www.ruby-doc.org/stdlib-1.9.3/libdoc/minitest/spec/rdoc/MiniTest/Expectations.html

- **must_equal** - Valores devem ser iguais. Para `Floats`, use `must_be_within_delta` explicada logo acima.

```
a.must_equal b
```

- **must_include** - A coleção deve incluir o objeto:

```
(0..10).must_include 5
```

- **must_match** - Deve "casar":

```
"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:

```
-> { puts "oi" }.must_output "oi\n"
=> true
-> { }.must_output "oi\n"
1) Failure:
test_0004_should output(Test) [minitest.rb:20]:
In stdout.
```

- **must_raise** - Deve disparar uma `Exception`:

```
-> { 1+"um" }.must_raise TypeError
=> true
-> { 1+1 }.must_raise TypeError
1) Failure:
test_0005_should raises an exception(Test) [minitest.rb:25]:
TypeError expected but nothing was raised.
```

- **must_respond_to** - Deve responder à um determinado método:

```
"oi".must_respond_to :upcase
```

- **must_send** - Deve poder ser enviado determinado método com argumentos:

```
must_send ["eustáquio",:slice,3,3]
```

- **must_throw** - Deve disparar um `throw`:

```
->{ 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:

```
class A; end
class B < A; end

b = B.new
b.instance_of?(B)
=> true

b.instance_of?(A)
=> false
b.kind_of?(B)
=> true
```

```
b.kind_of?(A)
=> true

A === b
=> true

B === b
true
```

Dá para perceber que `===`, para classes, é um *alias* de `kind_of?`.

13.5 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` [4], que automatizam esse processo. Podemos instalar as seguintes *gems* para utilizar `Guard` e `Minitest`:

```
$ gem install guard guard-minitest
```

Após isso, podemos executar:

```
$ guard init minitest
```

Deixar o arquivo `Guardfile` criado dessa maneira:

```
guard :minitest do
  watch(%r{^spec/(.*)_spec\.rb$})
  watch(%r{^(.+)\.rb$}) { |m| "spec/#{m[1]}_spec.rb" }
end
```

Código 238: Guardfile

Criar um diretório chamado `spec` (viram ele referenciado ali em cima?) com arquivos chamados `*_spec.rb` (também viram a máscara `*_spec.rb` ali?), copiar o arquivo `calc_spec3.rb` para `spec/calc_spec.rb` e finalmente rodar o comando `guard`:

```
$ guard
...
22:38:18 - INFO - Guard is now watching
[1] guard(main)>
```

Os testes encontrados vão ser avaliados sempre que algum arquivo com a extensão `.rb` no diretório corrente ou algum arquivo com o nome `*_spec.rb` for alterado. Note que a configuração do `Guardfile` procura saber qual é o teste para ser rodado através do nome do arquivo `.rb` modificado, inserindo `_spec` no final do nome dele.

[4] https://github.com/guard/guard

Capítulo 14

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.

14.1 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 o comando abaixo e responder algumas questões que nos são apresentadas da maneira que acharmos melhor (ficando a recomendação de responder `minitest` quando perguntado sobre testes):

```
$ bundle gem portnum
Creating gem 'portnum'...
```

> Quando rodar o comando, vai ser perguntado se queremos algum mecanismo de *continuous integration* para a *gem*. Podemos responder com `none` agora. Logo após, se queremos adicionar o `rubocop`. Podemos por enquanto responder com `n`, "não".

Esse comando gera um diretório chamado `portnum`, com a seguinte estrutura de diretório/arquivos, inclusive já dentro de um repositório do `Git`:

```
bin
Gemfile
.git
.gitignore
lib
portnum.gemspec
Rakefile
README.md
test
```

O ponto-chave é o arquivo `portnum.gemspec`:

```
# frozen_string_literal: true

require_relative "lib/portnum/version"

Gem::Specification.new do |spec|
  spec.name          = "portnum"
  spec.version       = Portnum::VERSION
  spec.authors       = ["Eustaquio Rangel"]
```

```
  spec.email         = ["taq@eustaquiorangel.com"]

  spec.summary       = "TODO: Write a short summary, because RubyGems requires one."
  spec.description   = "TODO: Write a longer description or delete this line."
  spec.homepage      = "TODO: Put your gem's website or public repo URL here."
  spec.required_ruby_version = Gem::Requirement.new(">= 2.3.0")

  spec.metadata["allowed_push_host"] = "TODO: Set to 'http://mygemserver.com'"

  spec.metadata["homepage_uri"] = spec.homepage
  spec.metadata["source_code_uri"] = "TODO: Put your gem's public repo URL here."
  spec.metadata["changelog_uri"] = "TODO: Put your gem's CHANGELOG.md URL here."

  # Specify which files should be added to the gem when it is released.
  # The `git ls-files -z` loads the files in the RubyGem that have been added into git.
  spec.files = Dir.chdir(File.expand_path(__dir__)) do
    `git ls-files -z`.split("\x0").reject { |f| f.match(%r{\A(?:test|spec|features)/}) }
  end
  spec.bindir        = "exe"
  spec.executables   = spec.files.grep(%r{\Aexe/}) { |f| File.basename(f) }
  spec.require_paths = ["lib"]

  # Uncomment to register a new dependency of your gem
  # spec.add_dependency "example-gem", "~> 1.0"

  # For more information and examples about making a new gem, checkout our
  # guide at: https://bundler.io/guides/creating_gem.html
end
```

Temos que preencher com os dados necessários:

```
# frozen_string_literal: true

require_relative "lib/portnum/version"

Gem::Specification.new do |spec|
  spec.name          = "portnum"
  spec.version       = Portnum::VERSION
  spec.authors       = ["Eustaquio Rangel"]
  spec.email         = ["taq@eustaquiorangel.com"]

  spec.summary       = "Aportuguesamento de números"
  spec.description   = "Adiciona os métodos par? e impar? na classe Numeric"
  spec.homepage      = "http://github.com/taq/portnum"
  spec.required_ruby_version = Gem::Requirement.new(">= 2.3.0")

  spec.metadata["allowed_push_host"] = "TODO: Set to 'http://mygemserver.com'"

  spec.metadata["homepage_uri"] = spec.homepage
  spec.metadata["source_code_uri"] = "http://github.com/taq/portnum"
  spec.metadata["changelog_uri"] = "http://github.com/taq/portnum/CHANGELOG.md"

  # Specify which files should be added to the gem when it is released.
  # The `git ls-files -z` loads the files in the RubyGem that have been added into git.
  spec.files = Dir.chdir(File.expand_path(__dir__)) do
    `git ls-files -z`.split("\x0").reject { |f| f.match(%r{\A(?:test|spec|features)/}) }
  end
  spec.bindir        = "exe"
  spec.executables   = spec.files.grep(%r{\Aexe/}) { |f| File.basename(f) }
  spec.require_paths = ["lib"]

  # Uncomment to register a new dependency of your gem
  # spec.add_dependency "example-gem", "~> 1.0"

  # For more information and examples about making a new gem, checkout our
  # guide at: https://bundler.io/guides/creating_gem.html
end
```

Código 239: Arquivo de criação de gems

Dentro do diretório `lib`, se encontram os seguintes arquivos:

```
$ ls -lah lib
portnum
portnum.rb

$ ls -lah lib/portnum
version.rb
```

Dentro do arquivo `version.rb`, temos:

```
$ cat lib/portnum/version.rb
# frozen_string_literal: true

module Portnum
  VERSION = "0.1.0"
end
```

Que vai definir o número de versão da nossa *gem*. Dentro do arquivo `portnum.rb`, temos:

```
$ cat lib/portnum.rb
```

```
require_relative "portnum/version"

module Portnum
  class Error < StandardError; end
  # Your code goes here...
end
```

Esse é o código que vai ser carregado quando a gem for requisitada. Vamos alterar a classe `Numeric` nesse arquivo (`lib/portnum.rb`), para implementar os nossos dois métodos:

```
# frozen_string_literal: true

require_relative "portnum/version"

class Numeric
  def par?
    self % 2 == 0
  end

  def impar?
    self % 2 != 0
  end
end
```

Código 240: Código da gem

14.2 Testando a gem

Antes de construir nossa *gem*, vamos criar alguns testes no diretório `test`, que deve estar criado:

```
# frozen_string_literal: true

require "test_helper"

class PortnumTest < Minitest::Test
  def test_that_it_has_a_version_number
    refute_nil ::Portnum::VERSION
  end

  def test_par
    assert_respond_to 1, :par?
  end

  def test_par_ok
    assert 2.par?
    assert !1.par?
  end

  def test_impar
    assert_respond_to 1, :impar?
  end

  def test_impar_ok
    assert 1.impar?
    assert !2.impar?
  end
end
```

Código 241: Código do teste da gem

Rodando os testes:

```
 rake test
Run options: --seed 22058

# Running:

.....

Finished in 0.001631s, 3066.3409 runs/s, 4292.8773 assertions/s.

5 runs, 7 assertions, 0 failures, 0 errors, 0 skips
```

14.3 Construindo a gem

Agora que verificamos que tudo está ok, vamos construir a nossa *gem*:

```
$ rake build
portnum 0.1.0 built to pkg/portnum-0.1.0.gem.

$ ls -lah pkg/
portnum-0.0.1.gem
```

Olha lá a nossa *gem*! Agora vamos instalá-la:

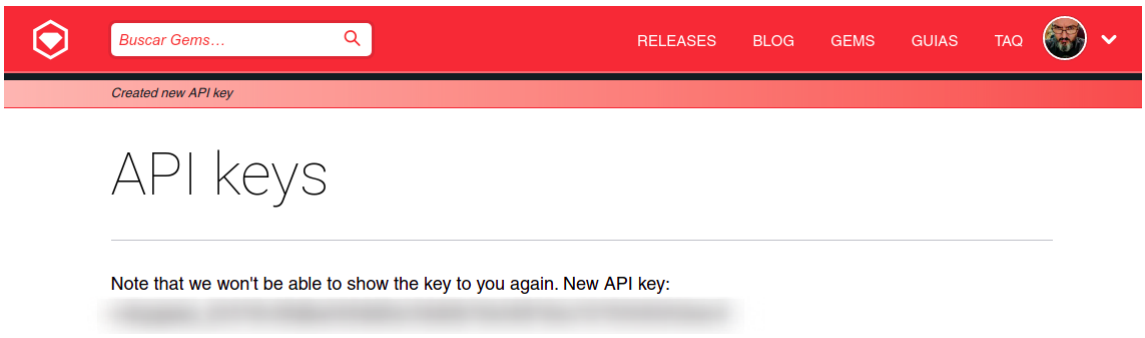
```
$ rake install
portnum 0.1.0 built to pkg/portnum-0.1.0.gem.
portnum (0.1.0) installed.
```

Testando se deu certo:

```
$ irb

require "portnum"
=> true
1.par?
=> false
1.impar?
=> true
```

14.4 Publicando a gem

Podemos publicar a gem facilmente para o RubyGems.org, que é o repositório oficial de *gems* para Ruby. Primeiro temos que criar uma conta lá, e indo em https://rubygems.org/profile/api_keys/new para gerar uma chave nova. Após gerar a chave, ela vai ser mostrada na tela e, **muito importante**, ela deve ser copiada e salva para um arquivo YAML em `/.gem/credentials`:



```
$ cat ~/.gem/credentials
---
:rubygems_api_key: nananinanao
```

Aí é só usar o comando `gem push`:

```
$ gem push portnum-0.0.1.gem
```

Se quisermos fazer os seguintes passos:

- Executar o `build`
- Criar uma tag no `git` e fazer um `push` para o repositório de código
- Publicar a *gem* no `RubyGems.org`

podemos utilizar:

```
$ rake release
portnum 0.0.1 built to pkg/portnum-0.0.1.gem
Tagged v0.0.1
...
```

Para ver todas as tasks que o Rake suporta:

```
$ rake -T
rake build            # Build portnum-0.1.0.gem into the pkg directory
rake clean            # Remove any temporary products
rake clobber          # Remove any generated files
rake install          # Build and install portnum-0.1.0.gem into system gems
rake install:local    # Build and install portnum-0.1.0.gem into system gems without network access
rake release[remote]  # Create tag v0.1.0 and build and push portnum-0.1.0.gem to Rubygems
rake test             # Run tests
```

14.5 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`:

```
$ gem unpack portnum
```

Ou, no caso de não ter as *gems* instaladas, utilizando a ferramenta `GNU tar`:

```
$ tar xvf portnum-0.0.1.gem data.tar.gz
$ tar tvf data.tar.gz
```

14.6 Assinando uma gem

Em razão de um problema de comprometimento do RubyGems em Janeiro de 2013, os autores de `gems` foram instruídos a assinarem as suas `gems` usando um certificado auto-assinado baseado com RSA, de forma que quando instaladas ou atualizadas, as `gems` possam ter a sua integridade verificada.

14.6.1 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:

```
$ gem cert --build <seu_email>
Passphrase for your Private Key:

Please repeat the passphrase for your Private Key:

Certificate: /home/taq/gem-public_cert.pem
Private Key: /home/taq/gem-private_key.pem
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`:

```
$ mkdir ~/.gemcert

$ mv -v gem-p* ~/.gemcert/
"gem-private_key.pem" -> "/home/taq/.gemcert/gem-private_key.pem"
"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.

14.6.2 Adaptando a gem para usar o certificado

Vamos pegar como exemplo uma `gem` que mantenho, a Traquitana. Para indicar que ela vai utilizar o meu certificado, vou inserir as seguintes linhas no final do arquivo `traquitana.gemspec`:

```
gem.signing_key = '/home/taq/.gemcert/gem-private_key.pem'
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 distruibuí-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):

```
$ cp ~/.gemspec/gem-public_cert.pem .
```

14.6.3 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:

```
$ rake build
Enter PEM pass phrase:
%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.

14.6.4 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:

```
$ gem update traquitana -P HighSecurity
Updating installed gems
Updating traquitana
Fetching: traquitana-0.0.23.gem (100%)
ERROR:  While executing gem ... (Gem::Security::Exception)
    root cert /CN=taq/DC=eustaquiorangel/DC=com is not trusted
```

Vamos dar uma olhada nos certificados que temos disponíveis:

```
$ 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.pem, de onde vamos fazer *download* para um arquivo local, importar o certificado e logo depois apagar o arquivo local:

```
$ curl https://raw.githubusercontent.com/taq/traquitana/master/gem-public_cert.pem > cert

$ gem cert --add cert
Added '/CN=taq/DC=eustaquiorangel/DC=com'

$ rm cert

$ gem cert --list
/CN=taq/DC=eustaquiorangel/DC=com
```

Com o certificado instalado, vamos tentar atualizar a `gem` novamente com a opção de verificação:

```
$ gem update traquitana -P HighSecurity
Updating installed gems
Updating traquitana
Successfully installed traquitana-0.0.23
Parsing documentation for traquitana-0.0.23
Installing ri documentation for traquitana-0.0.23
Installing darkfish documentation for traquitana-0.0.23
Done installing documentation for traquitana after 0 seconds
Gems updated: traquitana
```

Agora funcionou tudo de acordo!

> **Dica**
>
> Para utilizar a verificação com o Bundler, podemos utilizar a opção `-trust-policy HighSecurity`, que funciona da mesma forma demonstrada acima. Por exemplo:
>
> ```
> $ bundle install --trust-policy HighSecurity
> ```

Capítulo 15

Rake

Vimos no capítulo anterior uma ferramenta poderosíssima que utilizamos com bastante frequência no ecossistema Ruby: o Rake.

O `rake` foi inspirado no make, que é utilizado com frequência para automatizar tarefas, especialmente para compilar e gerar programas executáveis no mundo Unix. Sorte nossa que o `rake` é bem mais descomplicado e prático que o `make`, onde a geração de um `Makefile` mais completo (e complexo) demanda a utilização de outras ferramentas como o automake.

15.1 Definindo uma tarefa

Definir uma tarefa no `rake` é bem fácil. Primeiro, vamos precisar de um arquivo `Rakefile` (primo do `Makefile`), uma descrição e a definição da tarefa. Para o nosso exemplo, vamos fazer algumas tarefas para listar, criar o zip, e extrair os arquivos, mas utilizando os utilitários do sistema operacional (e não os meios que aprendemos em um capítulo anterior, para simplificar e focar aqui somente no `rake`).

Vamos criar os arquivos texto `1.txt`, `2.txt` e `3.txt`, com qualquer conteúdo, somente para utilizarmos novamente nesse capítulo:

```
$ ls *.txt
1.txt
2.txt
3.txt
```

Dando uma olhada no `Rakefile`:

```
desc "Lista os arquivos"
task :list do
  Dir.glob("*.txt") do |file|
    puts "encontrei o arquivo: #{file}"
  end
end
```

Código 242: Arquivo Rake inicial

Rodando o `rake`:

```
$ rake
rake aborted!
Don't know how to build task 'default' (see --tasks)
```

Ops, criamos uma *task* chamada `list` mas não especificamos qual seria a `task` *default* se rodarmos o `rake` sem uma *task* específica. Podemos indicar qual a *task* default utilizando `task default: <task>`:

```
task default: :list

desc "Lista os arquivos"
task :list do
  Dir.glob("*.txt") do |file|
    puts "encontrei o arquivo: #{file}"
  end
end
```

Código 243: Arquivo Rake com task default

Rodando novamente:

```
$ rake
encontrei o arquivo: 1.txt
encontrei o arquivo: 2.txt
encontrei o arquivo: 3.txt
```

Que é o mesmo comportamento que rodando com `rake list`:

```
$ rake list
encontrei o arquivo: 1.txt
encontrei o arquivo: 2.txt
encontrei o arquivo: 3.txt
```

A partir desse momento, já podemos listar quais são as tarefas definidas no `Rakefile` do diretório corrente, utilizando `rake -T`:

```
$ rake -T
rake list  # Lista os arquivos
```

15.2 Namespaces

Agora vamos imaginar que essa *task* `list`, como vimos aqui, lista os arquivos candidatos à compactação (que nesse caso, são apenas os arquivos `*.txt` que temos no diretório corrente), mas queremos também listar somente os arquivos já compactados, ou seja, os arquivos `.zip` presentes no diretório corrente. Seria outra *task* `list`, mas como evitar que uma *task* conflite com a outra? Da mesma forma que resolvemos isso com classes, utilizando *namespaces*:

```
task default: "files:list"

namespace :files do
  desc "Lista os arquivos candidatos à compactação"
  task :list do
    Dir.glob("*.txt") do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end

namespace :zip do
  desc "Lista os arquivos compactados"
  task :list do
    Dir.glob("*.zip") do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end
```

Código 244: Arquivo Rake com namespaces

Agora temos duas tarefas distintas:

```
$ rake -T
rake files:list  # Lista os arquivos candidatos à compactação
rake zip:list    # Lista os arquivos compactados
```

Uma diferença importante se não tivéssemos utilizado *namespaces* ali é que se definirmos outra tarefa com o mesmo nome de uma existente, **elas não se sobrepõem**, e sim a última é adicionada como uma continuação da anterior. Então, fiquem de olho nisso e organizem o seu código.

15.3 Tarefas dependentes

Vamos fazer uma tarefa agora para compactar os arquivos, apagando o arquivo `.zip` anterior se ele existir, definida na tarefa `clean`:

```
require "open3"

task default: "files:list"

filemask = "*.txt"
zipfile  = "rake.zip"

namespace :files do
  desc "Lista os arquivos candidatos à compactação"
  task :list do
    Dir.glob(filemask) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end

namespace :zip do
  desc "Lista os arquivos compactados"
  task :list do
    Dir.glob(zipfile) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end

  desc "Apaga o arquivo .zip anterior"
  task :clean do
    puts "Apagando o arquivo #{zipfile}, se existir ..."
    File.delete(zipfile) if File.exists?(zipfile)
  end

  desc "Cria o arquivo .zip"
  task build: :clean do
    puts "Criando o arquivo #{zipfile} ..."
    list = Dir.glob(filemask).sort.join(" ")
    puts "Adicionando os arquivos #{list} ..."
    stdin, stdout, stderr = Open3.popen3("zip #{zipfile} #{list}")
    error = stderr.read
    if error.size == 0
      puts "Arquivo criado com sucesso."
    else
      puts "Erro criando o arquivo: #{error}"
    end
  end
end
```

Código 245: Arquivo Rake com tarefas dependentes

Rodando a `task`:

```
$ rake zip:build
Apagando o arquivo rake.zip, se existir ...
Criando o arquivo rake.zip ...
Adicionando os arquivos 1.txt, 2.txt, 3.txt ...
Arquivo criado com sucesso.
```

15.4 Executando tarefas em outros programas

Podemos executar as tarefas em outros programas, como no `irb`:

```
$ irb
require "rake"
=> true

load "Rakefile"
=> true

Rake::Task["files:list"].invoke
encontrei o arquivo: 1.txt
encontrei o arquivo: 2.txt
encontrei o arquivo: 3.txt
=> [#<Proc:0x000000021b50d8@Rakefile:10>]
```

Reparem que no final é retornada uma `Proc`.

15.5 Arquivos diferentes

Até agora estamos executando todas as tarefas em um arquivo `Rakefile`, porém podemos ter vários arquivos `.rake` com código especificos, indicados na linha de comando, como por exemplo, `dependent.rake`:

```
require "open3"

task default: "files:list"

FILEMASK = "*.txt"
ZIPFILE  = "rake.zip"

namespace :files do
  desc "Lista os arquivos candidatos à compactação"
  task :list do
    Dir.glob(FILEMASK) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end

namespace :zip do
  desc "Lista os arquivos compactados"
  task :list do
    Dir.glob(ZIPFILE) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end

  desc "Apaga o arquivo .zip anterior"
  task :clean do
    puts "Apagando o arquivo #{ZIPFILE}, se existir ..."
    File.delete(ZIPFILE) if File.exists?(ZIPFILE)
  end

  desc "Cria o arquivo .zip"
  task build: :clean do
    puts "Criando o arquivo #{ZIPFILE} ..."
    list = Dir.glob(FILEMASK).sort.join(", ")
    puts "Adicionando os arquivos #{list} ..."
    stdin, stdout, stderr = Open3.popen3("zip #{ZIPFILE} #{list}")
    error = stderr.read
    if error.size == 0
      puts "Arquivo criado com sucesso."
    else
      puts "Erro criando o arquivo: #{error}"
    end
  end
end
```

Código 246: Arquivo Rake com nome fora do padrão

O que nos dá comportamento similar:

```
$ rake -f dependent.rake -T
rake files:list  # Lista os arquivos candidatos à compactação
rake zip:build   # Cria o arquivo .zip
rake zip:clean   # Apaga o arquivo .zip anterior
rake zip:list    # Lista os arquivos compactados

$ rake -f dependent.rake zip:build
Apagando o arquivo rake.zip, se existir ...
Criando o arquivo rake.zip ...
Adicionando os arquivos 1.txt, 2.txt, 3.txt ...
Arquivo criado com sucesso.
```

15.6 Tarefas com nomes de arquivo

Podemos definir uma *task* de arquivo, que somente vai ser executada se o arquivo não existir. Vamos criar uma chamada `rake.zip`, que vai executar, através de `invoke`, como vimos acima, a *task* `build`:

```
require "open3"

task default: "files:list"
filemask = "*.txt"
zipfile  = "rake.zip"

namespace :files do
  desc "Lista os arquivos candidatos à compactação"
  task :list do
    Dir.glob(filemask) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end

namespace :zip do
  desc "Lista os arquivos compactados"
  task :list do
    Dir.glob(zipfile) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end

  desc "Apaga o arquivo .zip anterior"
  task :clean do
    puts "Apagando o arquivo #{zipfile}, se existir ..."
    File.delete(zipfile) if File.exists?(zipfile)
  end

  desc "Cria o arquivo .zip"
  task build: :clean do
    puts "Criando o arquivo #{zipfile} ..."
    list = Dir.glob(filemask).sort.join(", ")
    puts "Adicionando os arquivos #{list} ..."
    stdin, stdout, stderr = Open3.popen3("zip #{zipfile} #{list}")
    puts stderr.read.size == 0 ? "Arquivo criado com sucesso." : "Erro criando o arquivo: #{error}"
  end

  desc "Cria o arquivo rake.zip se não estiver criado"
  file "rake.zip" do
    Rake::Task["zip:build"].invoke
  end
end
```

Código 247: Arquivo Rake com tarefas dependentes

Apagando o arquivo, rodando e verificando que da segunda vez a *task* não foi executada:

```
$ rm rake.zip

$ rake rake.zip
Apagando o arquivo rake.zip, se existir ...
Criando o arquivo rake.zip ...
Adicionando os arquivos 1.txt, 2.txt, 3.txt ...
Arquivo criado com sucesso.

$ rake rake.zip
```

15.7 Tarefas com listas de arquivos

Vimos que utilizamos `Dir.glob` para pegar a lista de arquivos, mas o próprio `rake` tem um método para selecionar e lidar com arquivos e nome de arquivos. Vamos adicionar alguns arquivos `*.txt` com nomes de letras (`a.txt`, `b.txt`, etc) e reescrever nosso `Rakefile` para:

```
require "open3"

task default: "files:list"
filemask = "*.txt"
zipfile  = "rake.zip"

namespace :files do
  desc "Lista os arquivos candidatos à compactação"
  task :list do
    Dir.glob(filemask) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end

namespace :zip do
  desc "Lista os arquivos compactados"
  task :list do
    Dir.glob(zipfile) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end

  desc "Apaga o arquivo .zip anterior"
  task :clean do
    puts "Apagando o arquivo #{zipfile}, se existir ..."
    File.delete(zipfile) if File.exists?(zipfile)
  end

  desc "Cria o arquivo .zip"
  task build: :clean do
    puts "Criando o arquivo #{zipfile} ..."
    list = Rake::FileList[filemask]
    list.exclude(/\A[a-zA-Z]+/)
    list = list.sort.join(", ")

    puts "Adicionando os arquivos #{list} ..."
    stdin, stdout, stderr = Open3.popen3("zip #{zipfile} #{list}")
    puts stderr.read.size == 0 ? "Arquivo criado com sucesso." : "Erro criando o arquivo: #{error}"
  end

  desc "Cria o arquivo rake.zip se não estiver criado"
  file "rake.zip" do
    Rake::Task["zip:build"].invoke
  end
end
```

Código 248: Arquivo Rake com máscaras de arquivo próprias

Dessa forma pedimos para excluir os arquivos que começam com letras (e mantenha os restantes) e quando rodamos temos:

```
$ rake zip:build
Apagando o arquivo rake.zip, se existir ...
Criando o arquivo rake.zip ...
Adicionando os arquivos 1.txt, 2.txt, 3.txt ...
Arquivo criado com sucesso.
```

15.8 Regras

Podemos ter regras de construção definidas através de expressões regulares, onde vai ser enviado o valor que "casa"com a expressão, através de um objeto do tipo `Rake::FileTask`:

```
require "open3"
task default: "files:list"
filemask = "*.txt"
zipfile  = "rake.zip"

namespace :files do
  desc "Lista os arquivos candidatos à compactação"
  task :list do
    Dir.glob(filemask) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end
namespace :zip do
  desc "Lista os arquivos compactados"
  task :list do
    Dir.glob(zipfile) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
  desc "Apaga o arquivo .zip anterior"
  task :clean do
    puts "Apagando o arquivo #{zipfile}, se existir ..."
    File.delete(zipfile) if File.exists?(zipfile)
  end
  desc "Cria o arquivo .zip"
  task build: :clean do
    puts "Criando o arquivo #{zipfile} ..."
    list = Rake::FileList[filemask]
    list.exclude(/\A[a-zA-Z]+/)
    list = list.sort.join(", ")

    puts "Adicionando os arquivos #{list} ..."
    stdin, stdout, stderr = Open3.popen3("zip #{zipfile} #{list}")
    puts stderr.read.size == 0 ? "Arquivo criado com sucesso." :  "Erro criando o arquivo: #{error}"
  end
  desc "Cria o arquivo rake.zip se não estiver criado"
  file "rake.zip" do
    Rake::Task["zip:build"].invoke
  end
  desc "Cria o arquivo"
  rule ".zip" do |file|
    zipfile = file.name
    Rake::Task["zip:build"].invoke
  end
end
```

Código 249: Arquivo Rake com expressões regulares

Rodando:

```
$ rake teste1.zip
Apagando o arquivo teste1.zip, se existir ...
Criando o arquivo teste1.zip ...
Adicionando os arquivos 1.txt, 2.txt, 3.txt ...
Arquivo criado com sucesso.

$ rake teste2.zip
Apagando o arquivo teste2.zip, se existir ...
Criando o arquivo teste2.zip ...
Adicionando os arquivos 1.txt, 2.txt, 3.txt ...
Arquivo criado com sucesso.
```

15.9 Estendendo

Lembram-se que se definirmos uma tarefa com o mesmo nome todas elas são executadas? Também podemos deixar esse comportamento mais explícito com `enhance`, que vai ser executado no final da tarefa que foi estendida:

```
require "open3"
require "fileutils"

task default: "files:list"
filemask = "*.txt"
zipfile  = "rake.zip"

namespace :files do
  desc "Lista os arquivos candidatos à compactação"
  task :list do
    Dir.glob(filemask) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end
end

namespace :zip do
  desc "Lista os arquivos compactados"
  task :list do
    Dir.glob(zipfile) do |file|
      puts "encontrei o arquivo: #{file}"
    end
  end

  desc "Apaga o arquivo .zip anterior"
  task :clean do
    puts "Apagando o arquivo #{zipfile}, se existir ..."
    File.delete(zipfile) if File.exists?(zipfile)
  end

  desc "Cria o arquivo .zip"
  task build: :clean do
    puts "Criando o arquivo #{zipfile} ..."
    list = Rake::FileList[filemask]
    list.exclude(/\A[a-zA-Z]+/)
    list = list.sort.join(", ")

    puts "Adicionando os arquivos #{list} ..."
    stdin, stdout, stderr = Open3.popen3("zip #{zipfile} #{list}")
    puts stderr.read.size == 0 ? "Arquivo criado com sucesso." : "Erro criando o arquivo: #{error}"
  end

  desc "Cria o arquivo rake.zip se não estiver criado"
  file "rake.zip" do
    Rake::Task["zip:build"].invoke
  end

  desc "Cria o arquivo"
  rule ".zip" do |file|
    zipfile = file.name
    Rake::Task["zip:build"].invoke
  end

  Rake::Task["zip:build"].enhance do
    newfile = "rake.#{Time.now.strftime('%H%M%S')}.zip"
    puts "Renomeando para #{newfile} ..."
    FileUtils.mv zipfile, newfile
  end
end
```

Código 250: Arquivo Rake inicial

Rodando:

```
$ rake zip:build
Apagando o arquivo rake.zip, se existir ...
Criando o arquivo rake.zip ...
Adicionando os arquivos 1.txt, 2.txt, 3.txt ...
Arquivo criado com sucesso.
Renomeando para rake.160626.zip ...
```

Capítulo 16

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:

```
# 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
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  # Parâmetros obrigatórios para criar o carro
  # Não se esqueça de que todo carro vai ter os custos de:
  # * IPVA
  # * Seguro obrigatório
  # * Seguro
  # * Manutenção
  def initialize(marca,modelo,cor,tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  # Converte o carro em uma representação mais legível
  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end
```

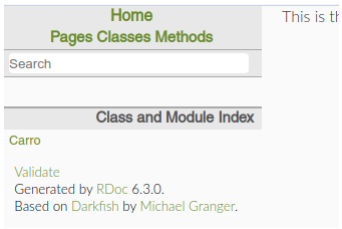
Código 251: Documentação da classe Carro

Agora vamos rodar o `rdoc` nesse arquivo:

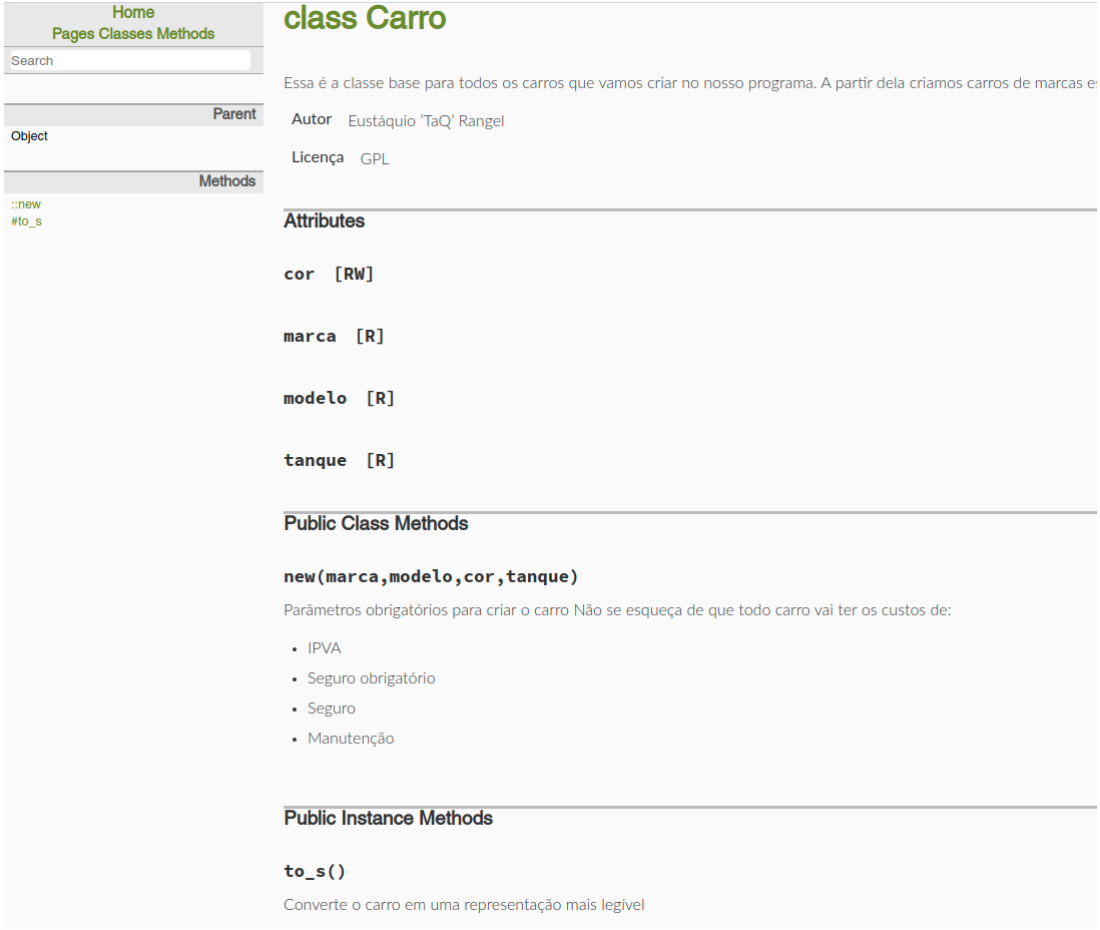
```
$ rdoc carro.rb
Parsing sources...
100% [ 1/ 1] carro.rb
Generating Darkfish format into doc...
Files: 1
Classes: 1
Modules: 0
Constants: 0
```

```
Attributes: 4
Methods: 2
(0 undocumented)
(0 undocumented)
(0 undocumented)
(4 undocumented)
(0 undocumented)

Total: 7 (4 undocumented)
42.86% documented
Elapsed: 0.1s
```

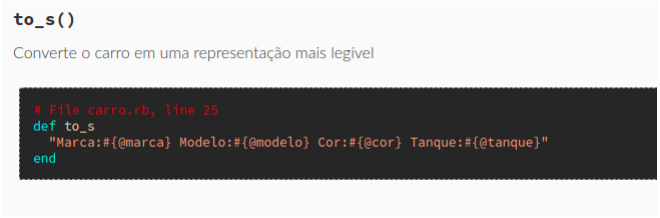
Isso vai produzir um diretório chamado `doc` abaixo do diretório atual, que vai conter um arquivo `index.html` com um conteúdo como esse:



Clicando no link da classe `Carro`, vamos ter algo como:



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:



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: = Primeiro nível == 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:

```ruby
# = 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:: http://eustaquiorangel.com
# Email:: mailto:naoteconto@eustaquiorangel.com
# Licença:: +GPL+ Clique aqui para ver mais[http://www.fsf.org]
#--
# Ei, ninguém deve ler isso.
#++
# Obrigado pela preferência.
class Carro
  attr_reader :marca, :modelo, :tanque
  attr_accessor :cor

  # Parâmetros obrigatórios para criar o carro
  # Não se esqueça de que todo carro vai ter os custos de:
  # * IPVA
  # * Seguro obrigatório
  # * Seguro
  # * Manutenção
  def initialize(marca,modelo,cor,tanque)
    @marca  = marca
    @modelo = modelo
    @cor    = cor
    @tanque = tanque
  end

  # Converte o carro em uma representação mais legível
  def to_s
    "Marca:#{@marca} Modelo:#{@modelo} Cor:#{@cor} Tanque:#{@tanque}"
  end
end

# Classe de um _vokinho_, derivada da classe Carro.
class Fusca < Carro
  def ipva
    false
  end
end
```

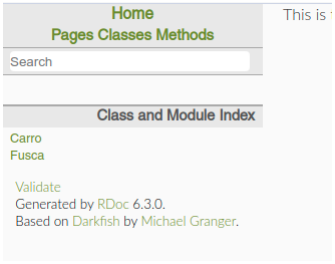
Código 252: Documentação de carros

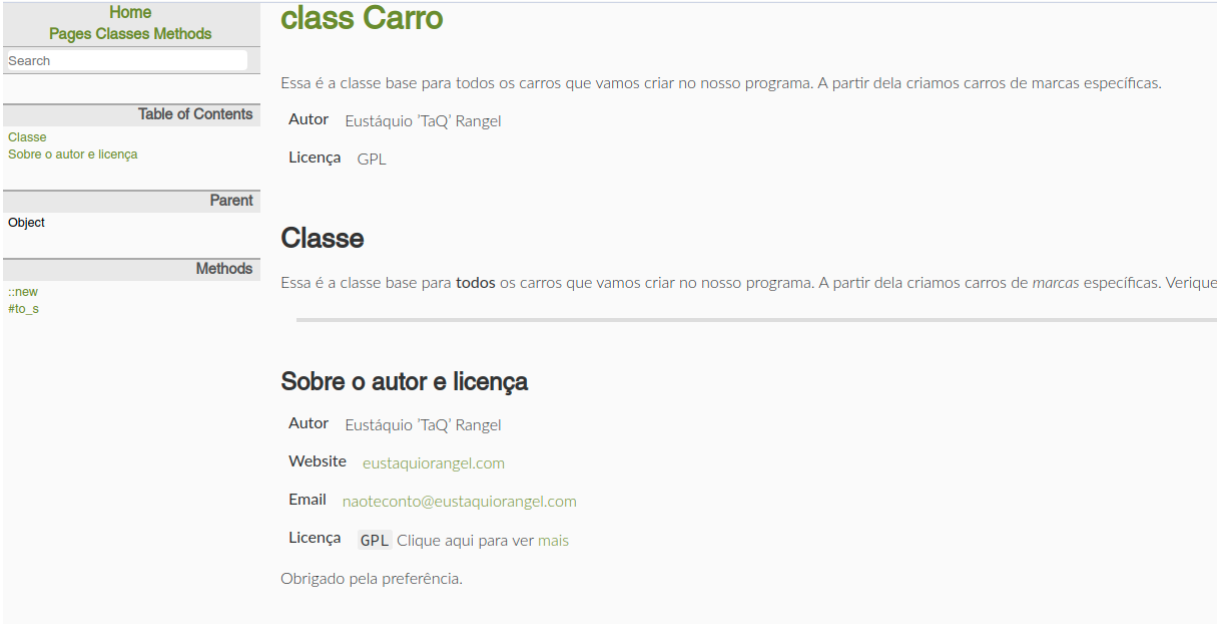
Rodando o `rdoc` (prestem atenção que agora não especifico o arquivo):

```
$ rdoc
Parsing sources...
100% [ 2/ 2] fusca.rb
Generating Darkfish format into /home/taq/git/curso-ruby-rails/code/rdoc/doc...
Files: 2
Classes: 2
Modules: 0
Constants: 0
Attributes: 4
Methods: 3
```

```
(0 undocumented)
(0 undocumented)
(0 undocumented)
(4 undocumented)
(1 undocumented)
Total:
9 (5 undocumented)
44.44% documented
Elapsed: 0.1s
```

Vamos ter um resultado como esse:





A partir da versão 2.0, o `Rdoc` entende Markdown. Para utilizar, devemos executar:

```
$ rdoc --markup markdown
```

E podemos deixar no diretório do projeto em um arquivo chamado `.doc_options`, para não ter que repetir toda vez, utilizando:

```
$ rdoc --markup markdown --write-options
```

Capítulo 17

Desafios

17.1 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:

```
x, y = 1, 2
y, x = x, y

x
=> 2

y
=> 1
```

17.2 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]`.

17.3 Desafio 3

Se criamos algo como:

```
v1 = "oi mundo"
v2 = Carro.new
v3 = 1
```

Isso significa que `v3` não vai apresentar a mensagem pois um `Fixnum` não aloca espaço na memória, que consequentemente não é processado pelo *garbage collector*.

17.4 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.

17.5 Desafio 5

Podemos atingir o mesmo comportamento usando `Hash` dessa forma:

```
str =<<FIM
texto para mostrar como podemos separar palavras do texto
para estatística de quantas vezes as palavras se repetem no
texto
FIM

p str.scan(/\w\p{Latin}+/).reduce(Hash.new(0)) { |memo, word| memo[word] += 1; memo }
```

17.6 Desafio 6

Seguindo a [URL da documentação do método documentação do método pack 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

17.7 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.

Lista de código-fonte

Índice Remissivo