Aprenda a Programar

Notas sobre a tradução em Português Brasileiro

Todo comunidade de programação é formada pelos amantes de computação, que aprendem linguagens novas no café-da-manhã, mas também uma uma grande parcela de pessoas que foram empolgadas por nós mas acabam esbarrando na barreira inicial do aprendizado. Felizmente autores como Chris Pine resolveram atacar este desafio e o resultado que temos é uma excelente material para programadores iniciantes em Ruby.

Eu, **Fabio Akita**, surgi com esta sugestão em Janeiro de 2008. Fiz o anúncio no meu blog e na lista rails-br e foi um movimento incrível: dezenas de voluntários se candidataram a ajudar. Graças a isso a tradução e revisão não durou uma semana! Fico extremamente satisfeito em ter essa amostra da comunidade se auto-ajudando. Espero ter a oportunidade de conduzir mais trabalhos desta natureza.

Meus agradecimentos e parabéns aos tradutores/revisores: Danilo Sato, Davi Vidal, Reginaldo Russinholi, Oliver Azevedo Barnes, Vitor Peres, Paulo Fernando Larini, Massimiliano Giroldi, Ricardo Yasuda, Lucas Húngaro, Anderson Leite. Agradeço também a todos que se voluntariaram, mas havia mais colaboradores que capítulos para traduzir e sobrou gente :-) Acho que é um bom sinal!

Agradecimentos também ao Júlio Santos Monteiro por dar uma "casa" oficial a este trabalho no web site http://aprendaaprogramar.rubyonrails.pro.br.

Esperamos que o resultado deste trabalho seja de grande valia tanto a estudantes quanto a qualquer um que queira ensinar outros a programar.

Um lugar para o futuro programador começar

Eu acho que tudo isso começou em 2002. Eu estava pensando em ensinar programação, e como uma grande linguagem como Ruby seria ótima para aprender a como programar. Quer dizer, nós estávamos todos excitados com Ruby pelo seu poder, elegância e por ser realmente divertido, mas me pareceu que ele também seria um excelente guia para aprender a programar.

Infelizmente, não havia muita documentação sobre Ruby destinada aos iniciantes naquela época. Alguns de nós, na comunidade, estávamos falando sobre como um tutorial "Ruby for the Nuby" era necessário, e, mais genericamente, um tutorial ensinando a programar, como um todo. Quanto mais eu pensava nisso, mais eu tinha a dizer (o que me surpreendeu um pouco). Até que alguém finalmente disse: "Chris, porque você não escreve um tutorial ao invés de ficar falando sobre isso?". Então eu o fiz.

E isso não foi muito bom. Eu tive todas essas boas idéias que eram boas *em teoria*, mas a real tarefa de fazer um grande tutorial para não-programadores foi muito mais desafiadora do que eu poderia prever. (Quer dizer, pareciam boas para mim, mas eu já sabia como programar.)

O que me salvou foi que eu fiz com que fosse fácil para as pessoas falarem comigo, e eu sempre tentei ajudar as pessoas quando elas empacavam. Quando eu via um monte de gente empacando em uma parte, eu a reescrevia. Isso deu muito trabalho, mas lentamente foi se tornando melhor e melhor.

Alguns anos depois, isso estava ficando realmente bom. :-) Tão bom, na verdade, que eu já estava pronto para anunciar sua finalização e partir para outra coisa. E justamente nesse instante houve a oportunidade de transformar esse tutorial em um livro. Uma vez que o básico já estava pronto, eu achei que não haveria maiores problemas. Eu apenas precisaria esclarecer umas coisas, adicionar alguns exercícios extras, talvez mais alguns exemplos, um pouquinho mais de capítulos, enviar ele para uns 50 revisores...

Isso me tomou outro ano, mas agora eu acho que está realmente *muito* bom, grande parte graças às centenas de boas almas que me ajudaram a escrever este livro.

O que está nesse site é o tutorial original, quase inalterado desde 2004. Para o melhor e mais atualizado, você pode querer dar uma olhada no livro.

Notas Para Professores

Há algumas normas de conduta que eu tentei seguir. Eu acho que elas tornam o processo de aprendizado muito mais suave; ensinar a programar já é difícil por si só. Se você está ensinando ou guiando alguém pelas vias hackers, essas idéias podem lhe ajudar também.

Primeiramente, eu tentei separar os conceitos o máximo possível, assim o estudante tem que aprender apenas um conceito de cada vez. Isso foi difícil no começo, mas um *pouco* mais fácil depois que eu peguei a prática. Algumas coisas devem ser faladas antes de outras, mas eu fiquei impressionado com quão pouca hierarquia de precedência realmente existe. Eventualmente, eu apenas tive que seguir uma ordem, e eu tentei arrumar as coisas de tal maneira que cada nova seção fosse motivada pela anterior.

Outro princípio que eu tinha em mente era de ensinar apenas uma maneira de fazer alguma coisa. Isso é um benefício óbvio em um tutorial para pessoas que nunca programaram antes. Por um motivo: é mais fácil aprender uma maneira de fazer uma coisa do que duas. Porém o benefício mais importante é que quanto menos coisas você ensinar a um novo programador, mais criativo e esperto ele tem que ser na programação. Já que muito da programação é resolução de problemas, torna-se crucial o encorajamento em todos os estágios possíveis.

Eu tentei traçar um paralelo entre os conceitos de programação com os conceitos que um novo programador já possui; para apresentar as idéias de uma maneira que o entendimento seja intuitivo, ao invés do tutorial despejar apenas informações. Programação Orientada a Objetos faz isso, por si só, muito bem. Eu fui capaz de me referir a "objetos" e diferentes "tipos de objetos" muito rapidamente nesse tutorial, soltando tais informações nos mais inocentes momentos. Eu não falei nada do tipo "tudo em Ruby é um objeto" ou "números e strings são tipos de objetos", porque essas coisas não dizem nada para um novo programador. Ao invés disso, eu vou falar sobre strings (e não sobre "objetos do tipo string"), e algumas vezes eu vou me referir a "objetos", apenas no sentido de "as coisas nesses programas". O fato de que todas essas *coisas* em Ruby *são* objetos fez com que esse tipo de inconsistência da minha parte funcionasse tão bem.

Sobretudo, eu procurei fugir do jargão desnecessário da OO, eu procurei ter certeza de que, se eles têm de aprender uma palavra, que aprendam a certa (Eu não quero que eles tenham de aprender em duplicidade, certo?). Então, eu chamei tudo de "strings", e não "texto". Métodos precisam ser chamados de alguma coisa, então eu os chamei de métodos.

A medida que os exercícios foram sendo concebidos, eu achei que estava com bons exercícios, mas você nunca pode colocar exercícios demais. Honestamente, eu acho que eu gastei quase metade do meu tempo apenas tentando fazer exercícios divertidos e interessantes. Exercícios entediantes apenas aniquilam qualquer desejo de programar, enquanto que o exercício perfeito cria aquela coceira no programador novo que ele não consegue ficar sem coçar. Resumindo, não gaste muito tempo tentando fazer exercícios bons.

Sobre o Tutorial Original

As páginas do tutorial (esta página, inclusive) são geradas por um grande programa em Ruby, claro. :-) Assim, elas possuem recursos elegantes. Por exemplo, todos os exemplos de código são realmente executados toda vez que você vê a página, e a saída dos mesmos é a saída que eles geram. Eu acredito que essa é a maneira mais fácil, melhor e, certamente, a mais legal de ter certeza que todo o código mostrado funciona *exatamente* como eu digo que funciona. Você não precisa se preocupar com a possibilidade de eu ter copiado a saída de um exemplo erroneamente ou esquecido de testar um código: tudo é testado na hora que você vê. Então, na seção de geradores de números aleatórios, você vai ver que os números mudam sempre... *lindo*. (Eu usei um truque parecido na hora de escrever o livro, mas é óbvio que isso é muito mais aparente aqui no tutorial.)

Sobre o Material Traduzido

O código original descrito por Chris Pine acima, era uma versão simples e implementada sobre CGI. Em total ritmo de 2008, eu mesmo (Fabio Akita) modifiquei esse código. Sem muitas modificações sobre o original, transportei o código para rodar sobre Rails 2.0.2.

Esse código está todo disponível no GitHub, neste endereço:

http://github.com/jmonteiro/aprendaaprogramar

Por motivos de performance, os códigos (que localmente são realmente executados em tempo real conforme Chris explicou) não são executados online, sendo uma cópia estática. Para ter todos os benefícios do programa em tempo real, baixe o código para rodar sobre Rails.

AGRADECIMENTOS

Finalizando, eu gostaria de agradecer a todos da lista ruby-talk por suas idéis e pelo encorajamento, aos meus maravilhosos revisores, por sua ajuda em fazer o livro muito melhor do que eu poderia fazer sozinho, especialmente à minha querida esposa, por ser minha principal revisora/testadora/porquinho-da-índia/musa, ao Matz, por ter criado essa fabulosa linguagem, e aos Pragmatic Programmers, por me falar sobre a linguagem—e, é claro, por publicar meu livro!

Se você notar qualquer erro ou falha de digitação, se tiver qualquer comentário ou sugestão, ou um bom exercício que eu possa incluir, por favor me avise (se você falar inglês) ou, se preferir falar em português, avise o Júlio Monteiro.

Aprenda a Programar

0. Iniciando

Quando você programa um computador, você tem que "falar" em uma língua que o seu computador entenda: uma linguagem de programação. Existem muitas e muitas linguagens por aí, e muitas são excelentes. Neste tutorial eu escolhi usar a minha favorita, *Ruby*.

Além de ser a minha favorita, Ruby também é a linguagem mais fácil que eu já vi (e eu já vi uma boa quantidade delas). Aliás, esta é a verdadeira razão pela qual estou escrevendo este tutorial: Eu não decidi escrever este tutorial e aí escolhi Ruby por ser minha favorita; ao invés disso, eu descobri que o Ruby era tão fácil que eu decidi que deveria haver um bom tutorial que a usasse voltado para iniciantes. Foi a simplicidade do Ruby que inspirou este tutorial, não o fato dela ser minha favorita. (Escrever um tutorial similar usando outra linguagem, C++ ou Java, teria tomado centenas e centenas de páginas.) Mas não pense que Ruby é uma linguagem para iniciantes só porque é fácil! Ela é uma linguagem poderosa, de nível profissional como poucas.

Quando você escreve algo em uma linguagem humana, o que é escrito é chamado de texto. Quando você escreve algo em uma linguagem de computador, o que é escrito é chamado de *código*. Eu incluí vários exemplos de código Ruby por todo este tutorial, a maioria deles programas completos que você pode rodar no seu próprio computador. Para deixar o código mais legível, eu colori partes dele com cores diferentes. (Por exemplo, números estão sempre em **`verde`**.) Qualquer coisa que você tiver que digitar estará sempre numa `caixa branca`, e qualquer coisa que o programa imprimir estará em uma **`caixa azul`**.

Se você encontrar algo que não entende, ou se você tiver uma pergunta que não foi respondida, tome nota e continue a ler! É bem possível que a resposta venha em um capítulo mais adiante. Porém, se sua pergunta não for respondida até o último capítulo, eu lhe mostrarei onde você pode ir para perguntar. Existem muitas pessoas maravilhosas por aí mais que dispostas a ajudar; você só precisa saber onde elas estão.

Mas primeiro nós precisamos baixar e instalar o Ruby no nosso computador.

Instalação no Windows

A instalação do Ruby no Windows é muito fácil. Primeiro, você precisa baixar o Instalador Ruby. Pode haver mais de versão para escolher; este tutorial usa a versão 1.8.7, então assegure-se de que o que você baixar seja ao menos tão recente quanto ela. (Eu pegaria a última versão disponível.) Então simplesmente rode o programa de instalação. Ele perguntará onde você gostaria de instalar o Ruby. A não ser que você tenha uma boa razão para não fazer isso, eu instalaria no lugar recomendado.

Para programar, você precisa poder escrever programas e rodá-los. Para fazer isso, você vai precisar de um editor de texto e uma linha de comando.

O instalador do Ruby vem com um editor de texto adorável chamado SciTE (the Scintilla Text Editor). Você pode rodar o SciTE selecionando-o no menu Iniciar. Se você quiser que o seu código seja colorido como os exemplos deste tutorial, baixe estes arquivos e coloque-os na sua pasta SciTE (`c:/ruby/scite` se você escolher o local recomendado).

- Propriedades Globais
- Propriedades do Ruby

Seria também uma boa idéia criar um diretório em algum lugar para manter todos os seus programas. Tenha certeza que, quando você salvar um programa, esteja salvando neste diretório.

Para ir para sua linha de comando, selecione Prompt de Comando na pasta Acessórios do seu menu Iniciar. Você vai querer navegar para o diretório onde você mantém seus programas. Digitar `cd ..` levará você para o diretório anterior, e `cd nome_do_diretorio` colocará você dentro do diretório chamado `nome_do_diretorio`. Para ver todos seus diretórios dentro do diretório atual, digite `dir /ad`.

E é isto! Você está pronto para aprender a programar.

Instalação para Macintosh

Se você tiver um Mac OS X 10.2 (Jaguar), então você já tem Ruby no seu sistema! O que poderia ser mais fácil? Infelizmente, eu não acho que você possa usar Ruby no Mac OS X 10.1 e versões anteriores.

Para programar, você precisa ser capaz de escrever programas e executá-los. Para fazer isto, você precisará de um editor de textos e uma linha de comando.

Sua linha de comando está acessível através da aplicação Terminal (encontrada em Aplicações/Utilitários).

Para um editor de textos, você pode usar qualquer um com que você esteja familiarizado ou se sinta confortável usando. Se você usa TextEdit, entretanto, tenha certeza que você está salvando seus programas como somente-texto! Caso contrário seus programas *não funcionarão*. Outras opções para programar são emacs, vi, e pico, que estão todos acessíveis via linha de comando.

E é isto! Você está pronto para aprender a programar.

INSTALAÇÃO EM LINUX

Primeiro, vale a pena checar se você já tem Ruby instalado. Digite `which ruby`. Se este comando responder algo como **`/usr/bin/which: no ruby in (...)`** , então você precisa fazer o download do Ruby, caso contrário veja que versão do Ruby você possui com `ruby -v`. Se for mais velha do que a última versão estável na página de download acima, pode ser bom atualizá-lo.

Se você for o usuário root, então você provavelmente não precisa de qualquer instrução para instalar o Ruby. Se não for, você poderia pedir ao seu administrador de sistema para instalá-lo para você. (Desta forma todos neste sistema poderiam usar Ruby.)

Caso contrário, você pode apenas instalá-lo de forma que apenas você possa usá-lo. Mova o arquivo baixado para um diretório temporário, como `$HOME/tmp`. Se o nome do arquivo for `ruby-1.6.7.tar.gz`, você pode abri-lo com `tar zxvf ruby-1.6.7.tar.gz`. Mude do diretório atual para o diretório que acabou de ser criado (neste exemplo, `cd ruby-1.6.7`).

Configure sua instalação digitando `./configure --prefix=$HOME`). Depois digite `make`, que construirá seu interpretador Ruby. Isto pode levar alguns minutos. Após isto ter terminado, digite `make install` para instalá-lo.

Em seguida, você vai querer adicionar `$HOME/bin` para seu caminho de busca de comandos à variável de ambiente PATH, editando seu arquivo `$HOME/.bashrc`. (Você pode ter que se deslogar e logar novamente para que isto surta efeito.) Após ter feito isto, teste sua instalação: `ruby -v`. Se mostrar a você qual a versão do Ruby que você tem, você pode agora remover os arquivos em `$HOME/tmp` (ou onde quer que você os colocou).

E é isto! Você está pronto para aprender a programar.

Aprenda a Programar

1. Números

Agora que você já arranjou tudo, vamos escrever um programa! Abra seu editor de texto favorito e digite o seguinte:

```
puts 1+2
```

Salve seu programa (sim, isso é um programa!) como `calc.rb` (o **.rb** é o que normalmente colocamos no final de programas escritos em Ruby). Agora rode o seu programa digitando `ruby calc.rb` na linha de comando. Ele deve ter posto `3` na sua tela. Viu como programar não é tão difícil?

Introdução ao puts

O que é então que está acontecendo no programa? Tenho certeza que você é capaz de adivinhar o quê `1+2` faz; nosso programa é praticamente a mesma coisa que:

```
puts 3
```

`puts` simplesmente escreve na tela tudo que vem depois dele.

Inteiro e Float

Na maioria das linguagens de programação (e não é diferente no Ruby) números sem pontos decimais são chamados de *inteiros*, e números com pontos decimais normalmente são chamados de *números de ponto-flutuante*, ou mais singelamente, *floats*.

Eis alguns inteiros:

```
5
-205
99999999999999999999999999
0
```

E aqui estão alguns floats:

```
54.321
0.001
-205.3884
0.0
```

Na prática, a maioria dos programas não usa floats; apenas inteiros. (Afinal, ninguém quer ler 7.4 emails, ou navegar 1.8 páginas, ou ouvir 5.24 músicas favoritas) Floats são usados mais freqüentemente para fins acadêmicos (experimentos de física e afins) e para gráficos 3D. Mesmo a maioria dos programas que lidam com dinheiro usam inteiros; eles só ficam contando as moedinhas!

Aritmética Simples

Até agora, temos tudo que é necessário para uma calculadora simples. (Calculadoras sempre usam floats, então se você quer que seu computador aja como uma calculadora, você também deve usar floats.) Para adição e subtração, usamos `+` e `-`, como vimos. Para multiplicação, usamos `*`, e para divisão usamos `/`. A maioria dos teclados possui essas teclas no teclado numérico. Se você tem teclado menor ou um laptop, você pode usar `Shift 8` e `/` (fica na mesma tecla que `?`). Vamos tentar expandir um pouco nosso calc.rb. Digite o seguinte e depois rode.

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

Isto é o que o programa retorna:

```
3.0
6.0
-3.0
4.5
```

(Os espaços no programa não são importantes; eles só deixam o código mais legível.) Bom, não foi lá muito surpreendente. Vamos tentar agora com inteiros.

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

Basicamente a mesma coisa, não é?

```
3
6
-3
4
```

Ahn... tirando aquele último ali! Quando você faz aritmética com inteiros, você recebe respostas em inteiros. Quando seu computador não sabe dar a resposta "certa", ele sempre arredonda para baixo. (Claro, **4** *é* a resposta certa em aritmética de inteiros para **9/2**; só pode não ser o que você estava esperando.)

Talvez você esteja se perguntado para que divisão de inteiros serve. Bem, vamos dizer que você vai ao cinema, mas só tem $ 9. Aqui em Portland, você pode ver um filme no Bagdad por 2 pilas. A quantos filmes você pode assistir lá? **9/2**... **4** filmes. 4.5 *não* é a resposta certa neste caso; eles não vão deixar você ver metade de um filme, ou metade de você ver um filme inteiro... algumas coisas não são divisíveis.

Agora experimente com alguns programas seus! Se você quiser escrever expressões mais complexas, você pode usar parênteses. Por exemplo:

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

```
5
-29802
```

Umas Coisinhas Para Tentar

Escreva um programa que lhe dê:

- quantas horas há em um ano?
- quantos minutos há em uma década?
- qual é a sua idade em segundos?
- quantos chocolates você pretende comer na vida?
  Aviso: *Esta parte do programa pode demorar um pouco para computar!*

Eis uma pergunta mais difícil:

- Se minha idade é de 1131 milhões de segundos, qual é minha idade em anos?

Quando você cansar de brincar com números, vamos dar uma olhada em algumas letras.

Aprenda a Programar

2. Letras

Então, nós já aprendemos tudo sobre números, mas e as letras? Palavras? Textos?

Nós nos referimos a grupos de letras em um programa como *strings* (Você pode pensar em letras impressas juntas ao longo de um banner). Para ficar mais fácil de entender quais partes do código são strings, Eu vou colorir as strings em **vermelho**. Aqui tem alguns exemplos de strings:

```
'Olá.'
'Ruby rocks.'
'5 é meu número favorito... qual é o seu?'
'Snoopy diz #%^?&*@! quando alguém pisa no seu pé.'
'     '
''
```

Como você pode ver, strings podem ter pontuação, dígitos, símbolos e espaços... muito mais do que apenas letras. A última string não tem nada: nós a chamamos de *string vazia*.

Nós estávamos usando **puts** para imprimir os números; vamos tentar ele de novo com algumas strings:

```
puts 'Olá, mundo!'
puts ''
puts 'Até logo.'
```

```
Olá, mundo!

Até logo.
```

Isso funcionou bem. Agora, tente umas strings você mesmo.

Aritimética das String

Assim como você pode fazer aritimética com números, você também pode fazer aritimética com strings! Bem, uma parte dela... você pode adicionar strings, de qualquer forma. Vamos tentar adicionar duas strings e ver o que o **puts** faz.

```
puts 'Eu gosto de' + 'torta de maçã.'
```

```
Eu gosto detorta de maçã.
```

Ops! Eu esqueci de adicionar um espaço entre **'Eu gosto de'** e **'torta de maçã.'**. Espaços não fazem importância normalmente, mas eles fazem sentido dentro de strings. (É verdade o que dizem: computadores não fazem o que você *quer* que eles façam, apenas o que você *manda* eles fazerem). Vamos tentar de novo:

```
puts 'Eu gosto de ' + 'torta de maçã.'
puts 'Eu gosto de' + ' torta de maçã.'
```

```
Eu gosto de torta de maçã.
Eu gosto de torta de maçã.
```

(Como você pode ver, não importa em qual string eu adicione o espaço.)

Então você pode somar strings. Mas você pode, também, multiplicá-las! (Por um número, de qualquer forma). Veja isso:

```
puts 'piscar ' * 4
```

```
piscando os olhos dela
```

(Estou brincando... ele na verdade faz isso:)

```
piscar piscar piscar piscar
```

Se você parar para pensar, isso realmente faz sentido. Afinal, `7*3` realmente quer dizer `7+7+7`, então `'moo'*3` apenas significa `'moo'+'moo'+'moo'`.

12 VS '12'

Antes de irmos mais longe, nós devemos ter certeza de que entendemos a diferença entre *números* e *dígitos*. `12` é um número, mas `'12'` é uma string de dois dígitos.

Vamos brincar com isso um pouco:

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

```
24
1212
12  +  12
```

Que tal isso?

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

```
10
22222
2  *  5
```

Esses exemplos foram muito diretos. De qualquer forma, se você não for muito cauteloso quando misturar strings e números, você pode cair em...

PROBLEMAS

Nesse ponto, você já deve ter tentado algumas coisas que *não* funcionaram. Se não, aqui tem algumas:

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

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

Hmmm... Uma mensagem de erro. O problema é que você não pode, realmente, adicionar um número a uma string, ou multiplicar uma string por outra string. Isso não faz muito sentido como isso:

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

Uma coisa que você deve saber: você pode escrever `'porco'*5` em um programa, já que isso apenas quer dizer `5` conjuntos da string `'porco'`, todas adicionadas entre si. Entretanto, você *não* pode escrever `5*'porco'`, já que isso significa `'porco'` conjuntos do número `5`, o que é um pouco insano.

Finalmente, e que tal um programa que imprima `Isso é um apóstrofo: '` ? Nós podemos tentar isso:

```
puts 'Isso é um apóstrofo: ''
```

Bem, *aquilo* não vai funcionar; Eu nem vou tentar executar aquilo. O computador me disse que nós terminamos com uma string. (É por isso que é bom ter um editor de texto que tenha *realçador de sintaxe* para você). Então, como podemos fazer com que o computador saiba que nós queremos continuar dentro da string? Nós temos que *escapar* o apóstrofo, assim:

```
puts 'Isso é um apóstrofo: \''
```

```
Isso é um apóstrofo: '
```

A barra invertida é um caractere de escape. Em outras palavras, se você tem uma barra invertida seguida de um caractere, isso pode ser, algumas vezes, traduzido em um novo caractere. As únicas coisas que uma barra invertida escapa, porém, são o apóstrofo e a própria barra invertida (Se você pensar a respeito, caracteres de escape devem sempre escapar a si mesmos).) Acho que uns exemplos são bons agora:

```
puts 'Isso é um apóstrofo: \''
puts 'uma barra invertida no fim da string:  \\'
puts 'acima\\embaixo'
puts 'acima\embaixo'
```

```
Isso é um apóstrofo: '
uma barra invertida no fim da string:  \
acima\embaixo
acima\embaixo
```

Uma vez que a barra invertida *não escapa*'e', mas *escapa* a si mesma, as últimas duas strings são idênticas. Elas não se parecem no código, mas no seu computador elas são as mesmas.

Se você tiver outra dúvida, apenas continue lendo! Eu não posso responder a todas as questões *nesta* página.

Aprenda a Programar

3. Variáveis e Atribuições

Até agora, sempre que usamos **puts** numa string ou número, o que imprimimos some. O que eu quero dizer é, se quiséssemos imprimir algo duas vezes, teríamos que digitar duas vezes:

```
puts '...você pode dizer aquilo de novo...'
puts '...você pode dizer aquilo de novo...'
```

```
...você pode dizer aquilo de novo...
...você pode dizer aquilo de novo...
```

Seria legal se pudéssemos digitá-lo uma única vez e mantê-lo por perto... guardá-lo em algum lugar. Bom, nós podemos, é claro—caso contrário eu não teria tocado no assunto!

Para guardar a string na memória de seu computador, precisamos dar um nome a ela. Normalmente os programadores chamam esse processo de *atribuição*, e chamam os nomes de *variáveis*. A variável pode ser praticamente qualquer seqüência de letras e números, mas o primeiro caracter tem de ser uma letra minúscula. Vamos rodar o último programa de novo, mas desta vez eu darei à string o nome de **minhaString** (ainda que eu pudesse tê-la chamado de **str** ou **minhaStringzinha** ou **pedroPrimeiro**).

```
minhaString = '...você pode dizer aquilo de novo...'
puts minhaString
puts minhaString
```

```
...você pode dizer aquilo de novo...
...você pode dizer aquilo de novo...
```

Sempre que você tentou fazer algo com **minhaString**, o programa fez com **'...você pode dizer aquilo de novo'** no lugar. Você pode pensar na variável **minhaString** como "apontando para" a string **'...você pode dizer aquilo de novo...'**. Eis um exemplo um pouquinho mais interessante:

```
nome = 'Patricia Rosanna Jessica Mildred Oppenheimer'
puts 'Meu nome é ' + nome + '.'
puts 'Nossa!  ' + nome + ' é um nome bem longo!'
```

```
Meu nome é Patricia Rosanna Jessica Mildred Oppenheimer.
Nossa!  Patricia Rosanna Jessica Mildred Oppenheimer é um nome bem longo!
```

Assim como podemos *atribuir* um objeto a uma variável, podemos *reatribuir* um objeto diferente à mesma variável (e é por isso que nós as chamamos de variáveis: porque a coisa para a qual apontam pode variar).

```
compositor = 'Mozart'
puts compositor + ' era "sensa", na sua época.'

compositor = 'Beethoven'
puts 'Mas eu, pessoalmente, prefiro ' + compositor + '.'
```

```
Mozart era "sensa", na sua época.
Mas eu, pessoalmente, prefiro Beethoven.
```

Variáveis podem, é claro, apontar para qualquer tipo de objeto, não só strings:

```
var = 'só mais uma ' + 'string'
puts var

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

```
só mais uma string
15
```

Na verdade, variáveis podem apontar para qualquer coisa... que não outras variáveis. Então o que acontece se tentarmos?

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

puts ''

var1 = 'oito'
puts var1
puts var2
```

```
8
8

oito
8
```

Primeiro, quando tentamos apontar **var2** para **var1**, **var2** apontou para **8** (exatamente como **var1** apontava). Aí fizemos **var1** apontar para **'oito'**, mas como **var2** nunca apontou de verdade para **var1**, ela se mantém apontando para **8**.

Agora que temos variáveis, números e strings, vamos aprender como misturar todos eles!

Aprenda a Programar

4. Misturando tudo

Nós vimos alguns tipos diferentes de objetos (números e letras), e nós fizemos variáveis para apontar para elas; a coisa que queremos fazer a seguir é fazer com que todas se encaixem.

Nós vimos que se desejássemos que um programa imprimisse **25**, o código a seguir ***não*** funcionaria, porque você não pode somar números e strings:

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Parte do problema é que seu computador não sabe se você está querendo **7** (**2 + 5**) ou **25** (**'2' + '5'**).

Antes que possamos somar os dois, precisamos de alguma maneira de obter a versão em string de **var1**, ou então de obter a versão inteira de **var2**.

Conversions

Para obter a versão string de um objeto, simplesmente escrevemos **.to_s** depois dele:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
```

```
25
```

Similarmente, **to_i** fornece a versão inteira de um objeto, e **to_f** dá a versão float. Vejamos o que esses três método fazem (e *não* fazem) com mais detalhe:

```
var1 = 2
var2 = '5'

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

```
25
7
```

Note que, mesmo depois que obtivemos a versão string de **var1** por meio de **to_s**, **var1** sempre apontou para **2**, e nunca para **'2'**. A não ser que reatribuamos **var1** explicitamente (o que requer um sinal de **=**), ela vai apontar para **2** enquanto o programa estiver rodando.

Agora vamos tentar algumas conversões mais interessantes (e alguma apenas esquisitas):

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts ''
puts '5 é meu número favorito!'.to_i
```

```
puts 'Quem foi que te perguntou sobre o 5?'.to_i
puts 'Sua mãe.'.to_f
puts ''
puts 'stringuelingue'.to_s
puts 3.to_i
```

```
15.0
99.999
99

5
0
0.0

stringuelingue
3
```

É provável que tenha havido surpresas. A primeira é bem padrão e dá **15**. Depois disso, convertemos a string **'99.999'** para um float e para um inteiro. O float ficou como esperávamos; o inteiro, como sempre, foi arredondado para baixo.

Em seguida temos alguns exemplos de strings... atípicas convertidas em números. **to_i** ignora a primeira coisa que ela não entende, e o resto da string daquele ponto em diante. Então a primeira string foi convertida para **5**, mas as outras, já que começavam com letras, foram ignoradas por completo... então o computador só escolhe zero.

Por fim, vimos que nossas duas últimas conversas não fizeram nada, exatamente como esperávamos.

Uma Outra Olhadela em puts

Há alguma coisa estranha no nosso método favorito... Dê uma olhadinha:

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

```
20
20
20
```

Por que é que essas três imprimem a mesma coisa? Tudo bem, as duas últimas deveriam mesmo, já que **20.to_s** *é* **'20'**. Mas e a primeira, o inteiro **20**? Falando nisso, faz algum sentido escrever *o inteiro* 20? Quando você escreve um *2* e depois um *0* num papel, você está escrevendo uma string, não um inteiro. *O inteiro* 20 é o número de dedos que eu possuo, e não um *2* seguido de um *0*.

O nosso amigo **puts** carrega um grande segredo: antes de tentar escrever um objeto, **puts** usa **to_s** para obter a versão string do mesmo. Na verdade, o *s* em **puts** está lá representando *string*; **puts** na verdade significa *put string* (colocar string).

Isso pode não parecer muito animador agora, mas há muitos, *muitos* tipos de objetos em Ruby (você vai até aprender a fazer o seu próprio!), e é bom saber o que vai acontecer se você tentar usar **puts** num objeto muito estranho, como uma foto da sua avó, ou um arquivo de música ou algo assim. Mas isso vem depois...

Até lá, nós temos mais alguns métodos para você, e eles permitem que escrevamos um bando de programas divertidos...

Os Métodos gets e chomp

Se **puts** significa *colocar string* (N.T.: put significa "colocar"), acho que você pode adivinhar o que **gets** quer dizer (N.T.: get, em inglês, entre várias acepções, significa pegar, obter). E assim como **puts** sempre cospe strings, **gets** vai apenas obter strings. E de onde ele as pega?

De você! Tudo bem, do seu teclado. Já que seu teclado só produz strings, isso funciona muito bem. O que acontece, na verdade, é que **gets** fica lá esperando, lendo o que você digita até que você pressione `Enter`. Vamos experimentar:

```
puts gets
```

```
Tem um eco aqui?
Tem um eco aqui?
```

Claro, tudo que você digitar vai ser repetido para você. Rode algumas vezes e tente digitar coisas diferentes.

Agora podemos fazer programas interativos! Neste, digite o seu nome para que ele lhe saude:

```
puts 'Olá, qual é o seu nome?'
name = gets
puts 'Seu nome é ' + name + '?  Que nome bonito!'
puts 'Prazer em conhecê-lo, ' + name + '.  :)'
```

Eca! Eu acabei de rodá-lo—escrevi meu nome—e aconteceu isto:

```
Olá, qual é o seu nome?
Chris
Seu nome é Chris
?  Que nome bonito!
Prazer em conhecê-lo, Chris
.  :)
```

Hmmm... parece que quando eu digitei as letras `C`, `h`, `r`, `i`, `s`, e pressionei `Enter`, **`gets`** obteve todas as letras do meu nome *e* o `Enter`! Felizmente, existe um método exatamente pra esse tipo de coisa: **`chomp`**. Ele retira qualquer `Enter que esteja lá de bobeira no fim da sua string. Vamos testar aquele programa de novo, mas com` **`chomp`** `para ajudar:`

```
puts 'Olá, qual é o seu nome?'
name = gets.chomp
puts 'Seu nome é ' + name + '?  Que nome bonito!'
puts 'Prazer em conhecê-lo, ' + name + '.  :)'
```

```
Olá, qual é o seu nome?
Chris
Seu nome é Chris?  Que nome bonito!
Prazer em conhecê-lo, Chris.  :)
```

Muito melhor! Perceba que já que **`name`** aponta para **`gets.chomp`**, não temos que dizer **`name.chomp`**; **`name`** já foi **`mastiga`**do (N.T.: chomp, em Inglês é "mastigar").

UMAS COISINHAS PARA TENTAR

• Escreva um programa que peça o nome de uma pessoa, depois o sobrenome. Por fim, faça com que ele cumprimente a pessoa usando seu nome completo.

• Escreva um programa que pergunte pelo número favorito de uma pessoa. Some um ao número, e sugira o resultado como um número favorito *muito melhor* (tenha tato ao fazê-lo).

Assim que você terminar esses dois programas (e quaisquer outros que você queira tentar), aprenderemos mais métodos (e mais coisas sobre eles).

5. Mais Sobre Métodos

Até agora nós vimos uma porção de métodos diferentes, **puts** e **gets** dentre outros (***Teste Surpresa:*** *Liste todos os métodos que vimos até agora! Foram dez deles; a resposta está mais embaixo.*), mas nós não conversamos realmente sobre o que são métodos. Nós sabemos o que eles fazem, mas não sabemos o que eles são.

Mas, na verdade, isso *é* o que eles são: coisas que fazem coisas. Se objetos (como strings, inteiros e floats) são os substantivos na linguagem Ruby, os métodos são como os verbos. E, ao contrário do Português (em que há sujeitos indefinidos e outras construções esotéricas), você não pode ter um verbo sem um substantivo para *executar* o verbo. Mas mesmo o Português trata a ausência de um substantivo como exceção: por exemplo, contar o tempo não é algo que simplesmente acontece; um relógio (ou cronômetro, ou algo parecido) deve fazê-lo. Em Português diríamos: "O relógio conta o tempo". Em Ruby dizemos **relogio.tiquetaque** (presumindo que **relogio** é um objeto Ruby, claro). Programadores podem dizer que estamos "chamando o método **tiquetaque** do **relogio**," ou que "chamamos **tiquetaque** no **relogio**."

E então, você respondeu o quiz? Ótimo. Bem, tenho certeza que você lembrou dos métodos **puts**, **gets** e **chomp**, que acabamos de ver. Você também provavelmente lembrou dos métodos de conversão, **to_i**, **to_f** e **to_s**. No entanto, você descobriu os outros quatro? Pois não eram ninguém menos que nossos velhos amigos da matemática, **+**, **-**, ***** e **/**!

Como eu estava dizendo, todo método precisa de um objeto. Geralmente é fácil dizer qual objeto está executando o método: é aquilo que vem logo antes do ponto, como no nosso exemplo do **relogio.tiquetaque**, ou em **101.to_s**. Às vezes, no entanto, isso não é tão óbvio; como com os métodos aritméticos. A bem da verdade, **5 + 5** é realmente apenas um atalho para se escrever **5.+ 5**. Por exemplo:

```
puts 'olá '.+ 'mundo'
puts (10.* 9).+ 9
```

```
olá mundo
99
```

Não é muito bonito, então nós nunca mais vamos escrever desse jeito; no entanto, é importante entender o que *realmente* está acontecendo. (Na minha máquina, isso também me dá um *aviso*: **warning: parenthesize argument(s) for future version**. O código ainda rodou sem problemas, mas ele está me dizendo que está com problemas para descobrir o que eu quis dizer, e pedindo para usar mais parênteses no futuro). Isso também nos dá um entendimento mais profundo sobre por que podemos fazer **'porco'*5** mas não **5*'porco'**: **'porco'*5** está dizendo ao **'porco'** para se multiplicar, mas **5*'porco'** está pedindo ao **5** que se multiplique. **'porco'** sabe como fazer **5** cópias de si mesmo e juntá-las; no entanto, **5** vai ter muito mais dificuldade para fazer **'porco'** cópias de *si mesmo* e juntá-las.

E, claro, nós ainda temos o **puts** e o **gets** para explicar. Onde estão seus objetos? Em Português, você pode às vezes omitir o substantivo; por exemplo, se um vilão grita "Morra!", o substantivo implícito é a pessoa com quem ele está gritando. Em Ruby, se dissermos **puts 'ser ou não ser'**, o que eu realmente estou dizendo é **self.puts 'ser ou não ser'**. Então o que é **self**? É uma variável especial que aponta para o objeto onde você está. Nós nem sabemos como estar *em* um objeto ainda, mas até descobrirmos, nós estaremos sempre em um grande objeto que é... o programa inteiro! E para nossa sorte, o programa tem alguns métodos próprios, como **puts** e **gets**. Preste atenção:

```
naoAcreditoQueFizUmNomeDeVariavelTaoGrandeApenasParaGuardarUm3 = 3
puts naoAcreditoQueFizUmNomeDeVariavelTaoGrandeApenasParaGuardarUm3
self.puts naoAcreditoQueFizUmNomeDeVariavelTaoGrandeApenasParaGuardarUm3
```

```
3
3
```

Se você não acompanhou tudo o que aconteceu, não tem problema. A coisa importante para aprender disso tudo é que cada métodos está sendo executado pelo mesmo objeto, mesmo que ele não esteja na sua frente. Se você entender isso, então está preparado.

Métodos Elegantes da String

Vamos aprender alguns métodos divertidos da string. Você não precisa memorizar todos eles; você pode apenas olhar esta página novamente se esquecê-los. Eu só quero mostrar uma *pequena* parte do que as strings podem fazer. Na verdade, eu mesmo não lembro da metade dos métodos da string—mas não tem problema, pois existem ótimas referências na internet com todos os métodos da string listados e explicados. (Vou mostrar onde encontrá-los no final deste tutorial.) Pra falar a verdade, eu nem *quero* saber todos os métodos da string; é como saber todas as palavras do dicionário. Eu posso falar Português muito bem sem saber todas as palavras do dicionário... e esse não é exatamente o seu propósito? Para que você não *precise* saber tudo que está nele?

Então, nosso primeiro método da string é o **reverse**, que nos dá uma versão ao contrário da string:

```
var1 = 'pare'
var2 = 'radar'
var3 = 'Voce consegue pronunciar esta frase ao contrario?'

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

```
erap
radar
?oirartnoc oa esarf atse raicnunorp eugesnoc ecoV
pare
radar
Voce consegue pronunciar esta frase ao contrario?
```

Como você pode ver, **reverse** não inverte a string original; ela apenas faz uma nova versão ao contrário dela. É por isso que **var1** ainda é **'pare'** mesmo após a chamada a **reverse** em **var1**.

Outro método da string é **length**, que nos diz o número de caracteres (incluindo espaços) na string:

```
puts 'Qual o seu nome completo?'
nome = gets.chomp
puts 'Você sabia que seu nome possui ' + nome.length + ' caracteres, ' + nome + '?'
```

```
Qual o seu nome completo?
Christopher David Pine
#<TypeError: can't convert Fixnum into String>
```

Uh-oh! Algo deu errado, e parece que isso aconteceu após a linha **`nome = gets.chomp`**... Você enxerga o problema? Veja se consegue descobrir.

O problema está em **`length`**: ele lhe dá um número, mas queremos uma string. Fácil, vamos colocar um **`to_s`** (e cruzar os dedos):

```
puts 'Qual o seu nome completo?'
nome = gets.chomp
puts 'Você sabia que seu nome possui ' + nome.length.to_s + ' caracteres, ' + nome +
```

```
Qual o seu nome completo?
Christopher David Pine
Você sabia que seu nome possui 22 caracteres, Christopher David Pine?
```

Não, eu não sabia disso. **Nota:** este é o número de *caracteres* no meu nome, não o número de *letras* (conte-as). Eu acho que conseguimos escrever um programa que pergunta seu primeiro nome, nome do meio e sobrenome individualmente e some todos os tamanhos... Ei, por que você não faz isso? Vá em frente, eu espero.

Pronto? Bom! É um programa legal, não é? Depois de mais uns capítulos, entretanto, você vai ficar maravilhado com o que conseguirá fazer.

Existem alguns métodos da string que conseguem mudar a caixa (maiúsculas e minúsculas) da sua string. **`upcase`** muda todas as letras minúsculas para maiúsculas, e **`downcase`** muda todas as letras maiúsculas para minúsculas. **`swapcase`** troca a caixa de todas as letras da string e, finalmente, **`capitalize`** é parecido com **`downcase`**, exceto que ele troca o primeiro caractere para maiúsculo (se for uma letra).

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

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

Coisas bem simples. Como você pode ver na linha **`puts ' a'.capitalize`**, o método **`capitalize`** apenas deixa em maiúsculo o primeiro *caractere*, não a primeira *letra*. Também, como vimos anteriormente, durante todas essas chamadas de métodos, **`letras`** continuou inalterada. Eu não quero me alongar nesse ponto, mas é importante entender. Existem alguns métodos que *mudam* o objeto associado,

mas ainda não vimos nenhum, e nem iremos ver durante algum tempo.

O último método elegante da string que iremos ver é para formatação visual. O primeiro, **`center`**, adiciona espaços no começo e no fim da string para torná-la centralizada. No entanto, assim como você precisa dizer ao **`puts`** o que quer que seja impresso, e ao **`+`** o que quer que seja adicionado, você precisa dizer ao **`center`** a largura total da string a ser centralizada. Então se eu quiser centralizar as linhas de um poema, eu faria assim:

```
larguraDaLinha = 50
puts(                'Old Mother Hubbard'.center(larguraDaLinha))
puts(               'Sat in her cupboard'.center(larguraDaLinha))
puts(         'Eating her curds an whey,'.center(larguraDaLinha))
puts(          'When along came a spider'.center(larguraDaLinha))
puts(         'Which sat down beside her'.center(larguraDaLinha))
puts('And scared her poor shoe dog away.'.center(larguraDaLinha))
```

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

Hum... Eu não acho que essa rima é assim, mas sou muito preguiçoso para procurar. (Também, eu queria alinhar a parte do **`.center larguraDaLinha`**, por isso acrescentei espaços extra antes das strings. Isso é só por que acho que fica mais bonito assim. Programadores geralmente têm opiniões fortes sobre o que é bonito num programa, e eles geralmente discordam sobre o assunto. Quanto mais você programar, mais vai descobrir seu próprio estilo). Falando em ser preguiçoso, a preguiça nem sempre é algo ruim na programação. Por exemplo, viu como eu guardei a largura do poema numa variável **`larguraDaLinha`**? Fiz isso pois se quiser tornar o poema mais largo mais tarde, só precisarei mudar a primeira linha do programa, ao invés de todas as linhas que são centralizadas. Com um poema muito longo, isso poderia me poupar um bom tempo. Esse tipo de preguiça é na verdade uma virtude na programação.

Então, em relação à centralização... você deve ter percebido que não está tão bonito quanto um processador de texto teria feito. Se você realmente quer centralização perfeita (e talvez uma fonte melhor), então você deveria apenas usar um processador de texto! Ruby é uma ferramenta maravilhosa, mas nenhuma ferramenta é a ferramenta certa para *qualquer* trabalho.

Os outros dois métodos de formatação da string são **`ljust`** e **`rjust`**, que fazem o texto *justificado à esquerda* e *justificado à direita*. Eles são parecidos com o **`center`**, exceto que eliminam os espaços em branco da string do lado direito e esquerdo, respectivamente. Vamos ver os três em ação:

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

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

Umas Coisinhas Para Tentar

• Escreva um programa do Chefe Zangado. Ele deve perguntar o que você quer de forma rude. Qualquer que seja a sua resposta, o Chefe Zangado vai gritar de volta para você, e então despedi-lo. Por exemplo, se você digitar `Eu quero um aumento.`, ele deve gritar de volta
`O QUE VOCÊ QUER DIZER COM "EU QUERO UM AUMENTO."?!? VOCÊ ESTÁ DESPEDIDO!!`

• Eis aqui algo para você fazer para brincar um pouco mais com **`center`**, **`ljust`** e **`rjust`**: Escreva um programa que irá mostrar um Índice de forma que fique parecido com:

```
                Tabela de Conteúdo

Capítulo 1:  Números                    página 1
Capítulo 2:  Letras                    página 72
Capítulo 3:  Variáveis                página 118
```

Matemática Avançada

*(Esta seção é totalmente opcional. Ela assume um certo nível de conhecimento matemático. Se você não estiver interessado, você pode ir direto para o Controle de Fluxo sem nenhum problema. No entanto, uma breve passada pela seção de **Números Aleatórios** pode ser útil.)*

Não há tantos métodos nos números quanto nas strings (apesar de eu ainda não ter decorado todos). Aqui, iremos olhar o resto dos métodos de aritmética, um gerador de números aleatórios e o objeto **`Math`**, com seus métodos trigonométricos e transcendentais.

Mais Aritimética

Os outros dois métodos aritméticos são **`**`** (exponenciação) e **`%`** (módulo). Então se você quisesse dizer "cinco ao quadrado" em Ruby, você escreveria **`5**2`**. Você também pode usar floats para seu expoente, então se você quiser a raiz quadrada de 5, você pode escrever **`5**0.5`**. O método módulo lhe dá o resto da divisão por um número. Então, por exemplo, se eu divido 7 por 3, eu tenho 2 com resto 1. Vamos vê-los em ação num programa:

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

```
25
2.23606797749979
2
1
1
```

Pela última linha, aprendemos que um ano (não-bissexto) tem um certo número de semanas, mais um dia. Então se seu aniversário caiu numa terça-feira este ano, ele será numa quarta-feira no ano que vem. Você também pode usar floats com o método módulo. Basicamente, ele funciona da única maneira razoável que consegue... Mas eu vou deixar você brincar um pouco com isso.

Tem um último método para ser mencionado antes de vermos o gerador de números aleatórios: **abs**. Ele simplesmente pega o valor absoluto de um número:

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

```
3
3
```

NÚMEROS ALEATÓRIOS

Ruby vem com um gerador de números aleatórios bem legal. O método que escolhe aleatoriamente um número é o **rand**. Se você chamar **rand** simplesmente assim, você vai ter um float maior ou igual a **0.0** e menor que **1.0**. Se você der um inteiro (**5** por exemplo) para o **rand**, ele vai te dar um inteiro maior ou igual a **0** e menor que **5** (então são cinco números possíveis, de **0** até **4**).

Vamos ver o **rand** em ação. (Se você recarregar esta página, estes números vão mudar a cada vez. Você sabia que eu estou realmente rodando estes programas, não sabia?)

```
puts rand
puts rand
puts rand
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(1))
puts(rand(1))
puts(rand(1))
puts(rand(99999999999999999999999999999999999999999999999999999999999))
puts('O homem do tempo disse que existe '+rand(101).to_s+'% de chance de chover,')
puts('mas você não pode nunca confiar num homem do tempo.')
```

```
0.204053403578031
0.854762319848877
0.613182884872693
32
45
37
0
0
0
693593600670415234072050190787515000767809401035519256962 78
O homem do tempo disse que existe 85% de chance de chover,
mas você não pode nunca confiar num homem do tempo.
```

Perceba que eu usei **rand(101)** para conseguir os números de **0** até **100**, e que **rand(1)** sempre retorna **0**. Não entender o intervalo dos possíveis valores retornados é o maior erro que eu vejo as pessoas cometerem ao usar **rand**; mesmo programadores profissionais; mesmo em produtos terminados que você pode comprar na loja. Eu inclusive tive um tocador de CD que, se configurado para "Tocar Aleatoriamente", iria tocar todas as músicas menos a última... (Eu imagino o que aconteceria se eu colocasse um CD com apenas uma música nele?)

As vezes você pode querer que **rand** retorne os *mesmos* números aleatórios na mesma sequência em duas execuções diferentes do seu programa. (Por exemplo, uma vez eu estava usando números aleatórios gerados para criar um mundo gerado aleatoriamente num jogo de computador. Se eu encontrasse um mundo que eu

realmente gostasse, talvez eu quisesse jogar nele novamente, ou mandá-lo para um amigo). Para isso, você precisa configurar a *semente*, que você consegue fazer com **srand**. Desta forma:

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

```
24
35
36
58
70

24
35
36
58
70
```

Ele fará a mesma coisa sempre que você alimentá-lo com a mesma semente. Se você quer voltar a ter números diferentes (como acontece quando você nunca usa **srand**), então apenas chame **srand 0**. Isso alimenta o gerador com um número realmente estranho, usando (dentre outras coisas) a hora atual do seu computador, com precisão de milisegundos.

O Objeto Math

Finalmente, vamos olhar para o objeto **Math**. Vamos começar de uma vez:

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

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

A primeira coisa que você percebeu foi provavelmente a notação do **::**. Explicar o *operador de escopo* (que é o que ele é) está um pouco além do, uh... escopo deste tutorial. Não é nenhum trocadilho. Eu juro. Basta dizer que você pode usar **Math::PI** exatamente da forma como você espera.

Como você pode ver, **Math** tem todas as coisas que você esperaria que uma calculadora científica decente

tivesse. E, como sempre, os floats estão *realmente perto* de serem as respostas certas.

Então vamos entrar no fluxo!

Aprenda a Programar

6. Controle de Fluxo

Ahhhh, controle de fluxo. É aqui que tudo se junta. Ainda que este capítulo seja mais curto e fácil que o capítulo sobre métodos, ele vai abrir um mundo de possibilidades de programação. Após este capítulo, poderemos escrever programas realmente interativos; antes fizemos programas que *dizem* coisas diferentes dependendo do que você escreve, mas após este capítulo eles *farão* coisas diferentes, também. Todavia, temos que poder comparar objetos no nosso programa. Precisamos de...

Métodos de Comparação

Vamos ser rápidos por aqui para chegar até as **ramificações**, que é onde as coisas legais começam a acontecer. Para ver se um objeto é maior ou menor que outro, usamos os métodos **>** e **<**, assim:

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

```
false
true
```

Tudo em ordem. Do mesmo modo, podemos descobrir se um objeto é maior-ou-igual-que (ou menor-ou-igual-que) outro com os métodos **>=** e **<=**.

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

```
true
false
```

Por fim, podemos descobrir se dois objetos são iguais ou não usando **==** (que significa "estes objetos são iguais?") e **!=** (que significa "estes objetos são diferentes?"). É importante não confundir **=** com **==**. **=** serve para dizer a uma variável que aponte para um objeto (atribuição), e **==** é para fazer a pergunta: "Estes dois objetos são iguais?".

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

```
true
true
```

E é claro que também podemos comparar strings. Quando strings são comparadas, leva-se em conta seus ordenamentos lexicográficos, que, trocando em miúdos, significa a ordem delas no dicionário. **`cachorro`** vem antes de **`gato`** no dicionário, então:

```
puts 'cachorro' < 'gato'
```

```
true
```

Há um porém: os computadores costumam ordenar letras maiúsculas antes de minúsculas, como se viessem antes (é assim que guardam as letras em fontes, por exemplo: todas as letras maiúsculas primeiro, seguidas das minúsculas). Isso significa que o

computador vai pensar que `'Zoológico'` vem antes de `'abelha'`, então se você quiser descobrir qual palavra viria primeiro num dicionário de verdade, use **`downcase`** (ou **`upcase`** ou **`capitalize`**) em ambas as palavras antes de tentar compará-las.

Uma última observação antes de **Ramificações**: os métodos de comparação não estão nos dando as strings `'true'` e `'false'`; elas estão nos dando os objetos especiais `true` e `false` (claro, `true.to_s` nos dá `'true'`, que é como **`puts`** imprimiu `'true'`). `true` e `false` são usados o tempo todo em...

Ramificações (Branching)

"Ramificação" é um conceito simples, mas poderoso. Na verdade, é tão simples que aposto que nem tenho que explicá-lo; deixa eu mostrar para você:

```
puts 'Olá, qual é o seu nome?'
nome = gets.chomp
puts 'Olá, ' + nome + '.'
if nome == 'Chris'
  puts 'Que nome bonito!'
end
```

```
Olá, qual é o seu nome?
Chris
Olá, Chris.
Que nome bonito!
```

Mas se colocarmos um nome diferente...

```
Olá, qual é o seu nome?
Chewbacca
Olá, Chewbacca.
```

E isso é ramificar. Se o que vem depois de `if` (N.T.—se) é `true`, nós executamos o código entre `if` e `end`. Se o que vem depois de `if` é `false`, não executamos nada. Claro e simples.

Eu indentei o código entre `if` e `end` porque acho que fica mais fácil acompanhar as ramificações assim. Quase todos os programadores fazem isso, independente da linguagem em que estejam trabalhando. Pode não parecer muito útil neste exemplo pequeno, mas quando as coisas ficam mais complexas, faz uma baita diferença.

Muitas vezes gostaríamos que um programa fizesse uma coisa se uma expressão for `true`, e outra se for `false`. É para isso que serve `else`:

```
puts 'Eu sou um vidente. Diga-me seu nome:'
nome = gets.chomp
if nome == 'Chris'
  puts 'Vejo coisas maravilhosas no seu futuro.'
else
  puts 'Seu futuro é... Ó, Deus! Olha a hora!'
  puts 'Eu tenho que ir, mil perdões!'
end
```

```
Eu sou um vidente. Diga-me seu nome:
Chris
Vejo coisas maravilhosas no seu futuro.
```

Agora vamos tentar um nome diferente...

```
Eu sou um vidente. Diga-me seu nome:
Ringo
Seu futuro é... Ó, Deus! Olha a hora!
Eu tenho que ir, mil perdões!
```

Ramificar é como deparar com uma bifurcação no código: tomamos o caminho para as pessoas com o **`nome == 'Chris'`** ou `else`, tomamos o outro caminho?

E como os galhos de uma árvore, você pode ter ramificações que contêm suas próprias ramificações:

```
puts 'Olá, e bem-vindo à aula de Português.'
puts 'Meu nome é professora Hélia. Seu nome é...?'
nome = gets.chomp

if nome == nome.capitalize
  puts 'Por favor, sente-se, ' + nome + '.'
else
  puts nome + '?  Você quer dizer ' + nome.capitalize + ', não é?'
  puts 'Você não sabe nem escrever seu nome??'
  resposta = gets.chomp

  if resposta.downcase == 'sim'
    puts 'Hunf! Vá, sente-se!'
  else
    puts 'FORA!!!'
  end
end
```

```
Olá, e bem-vindo à aula de Português.
Meu nome é professora Hélia. Seu nome é...?
chris
chris?  Você quer dizer Chris, não é?
Você não sabe nem escrever seu nome??
sim
Hunf! Vá, sente-se!
```

Está bem, vou capitalizar...

```
Olá, e bem-vindo à aula de Português.
Meu nome é professora Hélia. Seu nome é...?
Chris
Por favor, sente-se, Chris.
```

Às vezes pode ficar confuso entender onde colocar os **if**s, **else**s e **end**s. O que eu faço é escrever o **end** *ao mesmo tempo* que escrevo o **if**. Então enquanto eu estava escrevendo o programa acima, ele estava primeiro assim:

```
puts 'Olá, e bem-vindo à aula de Português.'
puts 'Meu nome é professora Hélia. Seu nome é...?'
nome = gets.chomp

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

Aí eu preenchi com *comentários*, coisas no código que o computador irá ignorar:

```
puts 'Olá, e bem-vindo à aula de Português.'
puts 'Meu nome é professora Hélia. Seu nome é...?'
nome = gets.chomp

if nome == nome.capitalize
  #  Ela é cordial.
else
  #  Ela fica brava.
end
```

Qualquer coisa após um **#** é considerado um comentário (a não ser, é claro, que você esteja em uma string). Depois de preencher com comentários, substituí-os por código funcional. Algumas pessoas gostam de deixá-los no arquivo; particularmente, creio que código bem-escrito fala por si. Eu costumava escrever mais comentários, mas quanto mais "fluente" fico em Ruby, menos faço uso deles. Eles me distraem boa parte do tempo. É uma escolha individual; você vai encontrar o seu estilo (normalmente em constante evolução). Então meu próximo passo ficou assim:

```
puts 'Olá, e bem-vindo à aula de Português.'
puts 'Meu nome é professora Hélia. Seu nome é...?'
nome = gets.chomp

if nome == nome.capitalize
  puts 'Por favor, sente-se, ' + nome + '.'
else
  puts nome + '?  Você quer dizer ' + nome.capitalize + ', não é?'
  puts 'Você não sabe nem escrever seu nome??'
```

```
  resposta = gets.chomp

  if resposta.downcase == 'sim'
  else
  end
end
```

Mais uma vez escrevi **if**, **else** e **end** ao mesmo tempo. Realmente me ajuda a saber "onde estou" no código. Também faz com que o trabalho pareça mais fácil porque posso me concentrar em uma parte pequena, como preencher o código entre **if** e **else**. Uma outra vantagem de fazê-lo desta maneira é que o computador pode entender o programa em qualquer estágio. Qualquer uma das versões inacabadas do programa que eu lhe mostrei rodariam. Elas não estavam terminadas, mas eram programas funcionais. Desta maneira eu pude testar enquanto escrevia, o que me ajudou a ver como o programa progredia e o que precisava ser melhorado. Quando ele passou em todos os testes, eu soube que estava pronto!

Essas dicas vão ajudá-lo a escrever programas que se ramificam, mas também ajudam com outro tipo central de controle de fluxo:

Repetição (Looping)

Você vai querer com alguma freqüência que o computador faça a mesma coisa várias e várias vezes—afinal, é nisso que os computadores, em teoria, são bons.

Quando você manda o seu computador ficar repetindo algo, você também precisa dizê-lo quando parar. Computadores nunca se entediam, então se você não mandar o seu parar, ele não parará. Garantimos que isso não aconteça ao dizermos que ele deve repetir certas partes de um programa **while** (N.T.—enquanto) uma condição especificada for verdadeira. O funcionamento é bem parecido com o do **if**:

```
comando = ''

while comando != 'tchau'
  puts comando
  comando = gets.chomp
end

puts 'Volte logo!'
```

```
Olá?
Olá?
Oi!
Oi!
Muito prazer em conhecê-lo.
Muito prazer em conhecê-lo.
Ah... que amor!
Ah... que amor!
tchau
Volte logo!
```

E isso é um loop (você deve ter notado a linha em branco no começo da saída; ela vem do primeiro **puts**, antes do primeiro **gets**. Como você modificaria o programa para se livrar dessa primeira linha? Teste! Funcionou *exatamente* como o programa acima, fora aquela primeira linha em branco?).

Loops, ou repetições, permitem que você faça várias coisas interessantes, como sei que você pode imaginar. Mas também podem causar problemas se você cometer algum erro. E se o computador ficar preso num loop infinito? Se você acha que isso pode ter acontecido, é só segurar a tecla `Ctrl` e pressionar `C`.

Antes de começarmos a brincar com loops, vamos aprender algumas coisinhas para facilitar nossa vida.

Um Pouco de Lógica

Vamos dar uma olhada no nosso primeiro programa com ramificações. E se minha esposa chegasse em casa, visse o programa, tentasse usá-lo e ele não dissesse que nome bonito *ela* tem? Eu não gostaria de magoá-la (ou de dormir no sofá), então vamos reescrevê-lo:

```
puts 'Olá, qual é o seu nome?'
nome = gets.chomp
puts 'Olá, ' + nome + '.'
if nome == 'Chris'
```

```
  puts 'Que nome bonito!'
else
  if nome == 'Katy'
    puts 'Que nome bonito!'
  end
end
```

```
Olá, qual é o seu nome?
Katy
Olá, Katy.
Que nome bonito!
```

Bom, funciona... mas não é lá um programa muito bonito. E por quê? A melhor regra que eu aprendi sobre programação foi a regra *DRY*: *Don't Repeat Yourself* (N.T.—Não Se Repita). Eu poderia escrever um livro só sobre o quão boa ela é. No nosso caso, repetimos a linha **`Que nome bonito!`**. Por que é que isso é um problema? Bem, e se eu cometi um erro de digitação quando eu reescrevi? E se eu quisesse mudar de **`bonito`** para gracioso em ambas as linhas? Eu sou preguiçoso, lembra? Basicamente, se eu quero que meu programa faça a mesma coisa quando receber **`Chris`** ou **`Katy`**, então ele realmente deve fazer *a mesma coisa*:

```
puts 'Olá, qual é o seu nome?'
nome = gets.chomp
puts 'Olá, ' + nome + '.'
if (nome == 'Chris' or nome == 'Katy')
  puts 'Que nome bonito!'
end
```

```
Olá, qual é o seu nome?
Katy
Olá, Katy.
Que nome bonito!
```

Muito melhor. Para que funcionasse, usei **`or`**. Os outros *operadores lógicos* são **`and`** e **`not`**. É sempre bom usar parênteses ao trabalhar com eles. Vamos ver como funcionam:

```
euSouChris  = true
euSouRoxo = false
euAmoComida = true
euComoPedras = false

puts (euSouChris and euAmoComida)
puts (euAmoComida and euComoPedras)
puts (euSouRoxo and euAmoComida)
puts (euSouRoxo and euComoPedras)
puts
puts (euSouChris or euAmoComida)
puts (euAmoComida or euComoPedras)
puts (euSouRoxo or euAmoComida)
puts (euSouRoxo or euComoPedras)
puts
puts (not euSouRoxo)
puts (not euSouChris)
```

```
true
false
false
false

true
true
true
false

true
false
```

O único deles que pode enganá-lo é **`or`** (N.T.—ou). Em português, usa-se "ou" para dizer "um ou outro, mas não os dois". Por exemplo, sua mãe pode lhe dizer: "Para sobremesa você pode escolher torta ou bolo". Ela *não* deu a opção de escolher os dois! Um computador, por outro lado, entende **`or`** como "ou um ou outro, ou os dois" (outro jeito de colocar é "ao menos um destes é verdadeiro"). É por isso que computadores são mais legais que mães.

UMAS COISINHAS PARA TENTAR

• *"Um elefante incomoda muita gente..."* Escreva um programa que imprima a letra para o clássico das viagens de carro, com um limite de 100 elefantes.

• Escreva um programa Velha Surda. O que quer que você diga à velha (o que quer que você digite), ela tem que responder com **`QUE?! FALA MAIS ALTO!`** , a não ser que você grite (digite tudo em maiúsculas). Se você gritar, ela pode lhe ouvir (ou ao menos pensa que pode), e sempre responde **`NÃO, NÃO DESDE 1938!`** Para fazer seu programa ser realmente verossímil, faça a velha gritar um número diferente a cada vez; talvez qualquer ano aleatório entre 1930 e 1950 (a última parte é opcional, e ficaria muito mais fácil se você lesse a seção sobre o gerador de números randômicos do Ruby no capítulo sobre métodos). Você não pode parar de falar com a velha enquanto não gritar `TCHAU`.
Dica: *Não esqueça do* **`chomp`** *!* **`'TCHAU'`** com um enter não é a mesma coisa que **`'TCHAU'`** sem! ***Dica 2:*** *Tente pensar em que partes do programa as coisas acontecem repetidamente. Todas elas devem estar no seu loop* **`while`**.

• Melhore o seu programa Velha Surda: e se a velha não quiser que você vá embora? Quando você gritar `TCHAU`, ela pode fingir que não lhe ouve. Mude seu programa anterior para que você tenha que gritar `TCHAU` três vezes *em seqüência*. Teste bem o seu programa: se você gritar `TCHAU` três vezes, mas não em seqüência, você tem que continuar falando com a velha.

• Anos bissextos. Escreva um programa que pergunte um ano inicial e um ano final, e imprima com **`puts`** todos os anos bissextos entre eles (e os incluindo, se eles também forem bissextos). Anos bissextos são sempre divisíveis por quatro (como 1984 e 2004). Contudo, anos divisíveis por 100 *não* são bissextos (como 1800 e 1900) ***a não ser que*** sejam divisíveis por 400 (como 1600 e 2000, que foram de fato anos bissextos). *(Sim, é bem confuso, mas não tão confuso como ter dezembro no meio do inverno, que é o que aconteceria no fim).*

Quando você terminá-las, descanse um pouco! Você já aprendeu muitas coisas. Parabéns. Você está surpreso com a quantidade de coisas que se pode mandar o computador fazer? Mais alguns capítulos e você vai poder programar praticamente tudo. Sério mesmo! Veja só tudo que você pode fazer que não podia antes de aprender sobre loops e ramificações.

Agora vamos aprender sobre um novo tipo de objeto, que controla listas de outros objetos: arrays.

Aprenda a Programar

7. Arrays e Iteradores

Vamos escrever um programa que nos permita entrar com quantas palavras quisermos (uma por linha, até pressionarmos `Enter` em uma linha vazia), e depois mostre as palavras em ordem alfabética. Ok?

Então... primeiro nós iremos.. bem.. hum... Bom, poderíamos.. rsrsrs..

Certo, não sei se podemos fazer isso. Precisamos de uma forma de armazenar um número qualquer de palavras, e de podermos acessá-las sem que se misturem com as outras variáveis. Precisamos colocá-las num tipo de lista. Precisamos dos *arrays*.

Um array é apenas uma lista em seu computador. Cada item da lista se comporta como uma variável: você pode ver qual objeto um item está apontando, e você pode fazê-lo apontar para um outro objeto. Vamos dar uma olhada em alguns arrays:

```
[]
[5]
['Olá', 'Tchau']

sabor = 'baunilha'             #  isto não é um array, claro...
[89.9, sabor, [true, false]]   #  ...mas isto é.
```

Primeiro nós temos um array vazio, então um array contendo um único número, então um array contendo duas strings. Depois, temos uma atribuição simples; e aí um array contendo três objetos, sendo que o último é um outro array **`[true, false]`**. Lembre-se, variáveis não são objetos, então, nosso último array está realmente apontando a para um float, uma *string* e um array. Mesmo que nós mudássemos o valor de **`sabor`**, isso não mudaria o array.

Para nos ajudar a encontrar um objeto qualquer num array, cada item tem um número de indexação. Programadores (e, aliás, a maioria dos matemáticos) iniciam contando do zero, então, o primeiro item do array é o item zero. Veja como podemos referenciar os objetos em um array:

```
nomes = ['Ana', 'Maria', 'Cris']

puts nomes
puts nomes[0]
puts nomes[1]
puts nomes[2]
puts nomes[3]  #  Isto está fora do intervalo
```

```
Ana
Maria
Cris
Ana
Maria
Cris
nil
```

Vemos então, que **`puts nomes`** imprime cada nome do array **`nomes`**. Então usamos **`puts nomes[0]`** para imprimir o "primeiro" nome do array e **`puts nomes[1]`** para imprimir o "segundo"... Tenho certeza que parece confuso, mas você *deve* se acostumar com isso. Você deve realmente começar a *acreditar* que contar inicia do zero e parar de usar palavras como "primeiro" e "segundo". Se você for num rodízio de pizza, não fale sobre o "primeiro" pedaço; fale sobre o pedaço zero (e na sua cabeça pense **`pedaco[0]`**). Você tem 5 dedos na sua mão e seus números são 0, 1, 2, 3 e 4. Minha esposa e eu somos malabaristas. Quando fazemos malabares com 6 pinos, estamos equilibrando os pinos 0 a 5. Felizmente, em alguns meses, estaremos equilibrando o pino 6 (e portanto, equilibrando 7 pinos). Você saberá que conseguiu quando começar a usar o termo "zerésimo" :-) Sim, é uma palavra real.. Pergunte a um programador ou matemático..

Finalmente, nós tentamos **`puts nomes[3]`**, apenas veja o que aconteceria. Você estava esperando um erro? Algumas vezes quando você faz uma pergunta, sua pergunta não faz sentido (pelo menos para seu computador); é quando obtém um erro. Algumas vezes, entretanto, você pode fazer uma pergunta e a resposta é *nada*. O que está na posição três? Nada. O que é

nomes[3]? **nil**: A maneira Ruby de dizer "nada". **nil** é um objeto especial que basicamente significa "não é qualquer outro objeto."

Se toda esta numeração divertida de posições de array está confundindo você, não tema! Freqüentemente, nós podemos evitá-la completamente usando vários métodos de array, como este:

O Método each

each nos permite fazer algo (o que quer que nós desejemos) para **each** (cada em português) objeto pertencente ao array. Assim, se nós quiséssemos dizer algo bom sobre cada linguagem no array abaixo, nós faríamos isto:

```
linguagens = ['Português', 'Inglês', 'Ruby']

linguagens.each do |ling|
  puts 'Eu adoro ' + ling + '!'
  puts 'Você não?'
end

puts 'E vamos ouví-lo sobre C++!'
puts '...'
```

```
Eu adoro Português!
Você não?
Eu adoro Inglês!
Você não?
Eu adoro Ruby!
Você não?
E vamos ouví-lo sobre C++!
...
```

Então, o que aconteceu? Bem, nós fomos capazes de passar por todos os objetos no array sem usar nenhum número, e isto é muito bom. Traduzindo para o português, o programa acima seria algo como: Para **cada** objeto em **linguagens**, aponte a variável **ling** para o objeto e então **faça** (do em inglês) tudo que eu disser, até que você chegue ao **fim** (end em inglês). (Como você sabe, C++ é uma outra linguagem de programação. É muito mais difícil de aprender do que Ruby; normalmente, um programa C++ será muitas vezes maior do que um programa Ruby que faz a mesma coisa.)

Vocês poderiam estar pensando consigo mesmos, "Isto é muito parecido com os laços de repetição (loops) que nós aprendemos anteriormente." Sim, é similar. Uma diferença importante é que o método **each** é apenas: um método. **while** e **end** (como **do**, **if**, **else**, e todas as outras palavras em **azul**) não são métodos. Elas são parte fundamental da linguagem Ruby, como **=** e os parênteses; como os sinais de pontuações no português.

Mas não **each**; **each** é um apenas um outro método do array. Métodos como **each** que "atuam como" loops são freqüentemente chamados *iteradores*.

Uma coisa para notar sobre iteradores é que eles são sempre seguidos por **do**...**end**. **while** e **if** nunca têm um **do** perto deles; nós apenas usamos **do** com iteradores.

Aqui está um outro atraente iteradorzinho, mas não é um método de array... é um método de inteiros!

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

```
Hip-Hip-Urra!
Hip-Hip-Urra!
Hip-Hip-Urra!
```

Mais Métodos de Array

Então nós aprendemos **each**, mas existem muitos outros métodos de array... quase tantos quantos existem métodos de string! Na verdade, alguns deles (como **length**, **reverse**, **+**, e *****) funcionam da mesma forma que funcionam para strings, exceto que eles operam em posições de array ao invés de em letras de string. Outros, como **last** e **join**, são específicos para arrays. Ainda outros, como **push** e **pop**, na verdade modificam o array. E assim como com os métodos de string, você não tem que se lembrar de todos estes, desde que você possa se lembrar onde achar informações sobre eles (bem aqui).

Primeiro, vamos dar uma olhada em **to_s** e **join**. **join** funciona de forma semelhante a **to_s**, exceto que ele adiciona uma string entre os objetos do array. Vamos dar uma olhada:

```
comidas = ['feijoada', 'tapioca', 'bolo de fubá']

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

200.times do
  puts []
end
```

```
feijoada
tapioca
bolo de fubá

feijoadatapiocabolo de fubá

feijoada, tapioca, bolo de fubá

feijoada  :)  tapioca  :)  bolo de fubá  8)
```

Como você pode ver, **puts** trata os arrays diferentemente de outros objetos: ele apenas chama **puts** em cada um dos objetos no array. É por isso que chamar **puts** para um um array vazio 200 vezes não faz nada; o array não aponta para nada, assim não há nada para o **puts** mostrar (Fazer nada 200 vezes ainda é fazer nada). Tente chamar **puts** para um array contendo outros arrays; fez o que você esperava?

Você também notou que eu deixei uma string vazia quando eu quis **mostrar** uma linha em branco? Isto faz a mesma coisa.

Agora vamos dar uma olhada em **push**, **pop**, e **last**. Os métodos **push** e **pop** são de alguma forma opostos, assim como **+** e **-** são. **push** adiciona um objeto no fim do seu array, e **pop** remove o último objeto do array (e diz para você qual era este objeto). **last** é similar a **pop** em dizer para você o que está no fim do array, exceto que o **last** deixa o array em paz. Novamente, **push** e **pop** *realmente modificam o array*:

```
favoritos = []
favoritos.push 'azul e branco'
favoritos.push 'verde e amarelo'

puts favoritos[0]
puts favoritos.last
puts favoritos.length

puts favoritos.pop
puts favoritos
puts favoritos.length
```

```
azul e branco
verde e amarelo
2
verde e amarelo
azul e branco
1
```

Algumas Coisas para Tentar

• Escreva o programa que nós comentamos no início deste capítulo.
Dica: *Existe um adorável método de array que dará a você uma versão ordenada de um array:* **sort**. *Use-o!*

• Tente escrever o programa acima *sem* usar o método **sort**. Uma grande parte da programação é resolver problemas, assim pratique o máximo que você puder!

• Re-escreva seu programa de Tabela de Conteúdos (do capítulo métodos). Inicie o programa com um array que mantém todas as informações sobre sua Tabela de Conteúdos (nomes de capítulos, números de páginas, etc.). Então imprima na tela a informação do array em uma Tabela de Conteúdos, formatada de forma bem bonita.

Até o momento aprendemos bastante sobre um número de métodos diferentes. Agora é hora de aprender como fazer seus próprios métodos.

8. Escrevendo seus Próprios Métodos

Como vimos, repetições e iteradores nos permitem fazer a mesma coisa (rodar o mesmo código) várias e várias vezes. Porém, algumas vezes queremos fazer a mesma coisa um monte de vezes, mas de lugares diferentes do programa. Por exemplo, vamos supor que estejamos escrevendo um programa de questionário para um estudante de psicologia. Levando em conta os estudantes de psicologia que eu conheço e os questionários que eles me forneceram, seria algo parecido com isto:

```
puts 'Olá, e obrigado pelo seu tempo para me ajudar'
puts 'com essa pesquisa. Minha pesquisa é sobre'
puts 'como as pessoas se sentem com comida'
puts 'mexicana. Apenas pense sobre comida mexicana'
puts 'e tente responder, honestamente, cada questão'
puts 'com "sim" ou "não". Minha pesquisa não tem'
puts 'nada a ver com quem molha a cama.'
puts

#  Nós fazemos as perguntas, mas ignoramos as respostas.

boaResposta = false
while (not boaResposta)
  puts 'Você gosta de comer tacos?'
  resposta = gets.chomp.downcase
  if (resposta == 'sim' or resposta == 'não')
    boaResposta = true
  else
    puts 'Por favor, responda com "sim" ou "não".'
  end
end

boaResposta = false
while (not boaResposta)
  puts 'Você gosta de comer burritos?'
  resposta = gets.chomp.downcase
  if (resposta == 'sim' or resposta == 'não')
    boaResposta = true
  else
    puts 'Por favor, responda com "sim" ou "não".'
  end
end

#  Porém, nós prestamos atenção *nesta* questão.
boaResposta = false
while (not boaResposta)
  puts 'Você faz xixi na cama?'
  resposta = gets.chomp.downcase
  if (resposta == 'sim' or resposta == 'não')
    boaResposta = true
    if resposta == 'sim'
      molhaCama = true
    else
      molhaCama = false
    end
  else
    puts 'Por favor, responda com "sim" ou "não".'
  end
```

```
end

boaResposta = false
while (not boaResposta)
  puts 'Você gosta de comer chimichangas?'
  resposta = gets.chomp.downcase
  if (resposta == 'sim' or resposta == 'não')
    boaResposta = true
  else
    puts 'Por favor, responda com "sim" ou "não".'
  end
end

puts 'Apenas mais algumas perguntas...'

boaResposta = false
while (not boaResposta)
  puts 'Você gosta de comer sopapillas?'
  resposta = gets.chomp.downcase
  if (resposta == 'sim' or resposta == 'não')
    boaResposta = true
  else
    puts 'Por favor, responda com "sim" ou "não".'
  end
end

#  Faça mais um monte de perguntas sobre comida
#  mexicana.

puts
puts 'QUESTIONÁRIO:'
puts 'Obrigado por dispender seu tempo ao nos ajudar'
puts 'com nossa pesquisa. Na verdade, essa pesquisa'
puts 'não tem nada a ver com comida mexicana.'
puts 'Mas é uma pesquisa sobre quem molha a cama.'
puts 'As comidas mexicanas estavam lá apenas para'
puts 'baixar sua guarda na esperança de fazer você'
puts 'responder mais honestamente. Obrigado novamente.'
puts
puts molhaCama
```

```
Olá, e obrigado pelo seu tempo para me ajudar
com essa pesquisa. Minha pesquisa é sobre
como as pessoas se sentem com comida
mexicana. Apenas pense sobre comida mexicana
e tente responder, honestamente, cada questão
com "sim" ou "não". Minha pesquisa não tem
nada a ver com quem molha a cama.

Você gosta de comer tacos?
sim
Você gosta de comer burritos?
sim
Você faz xixi na cama?
de jeito nenhum!
Por favor, responda com "sim" ou "não".
Você faz xixi na cama?
NÃO
Por favor, responda com "sim" ou "não".
Você faz xixi na cama?
sim
Você gosta de comer chimichangas?
sim
Apenas mais algumas perguntas...
Você gosta de comer sopapillas?
sim

QUESTIONÁRIO:
Obrigado por dispender seu tempo ao nos ajudar
com nossa pesquisa. Na verdade, essa pesquisa
não tem nada a ver com comida mexicana.
```

```
Mas é uma pesquisa sobre quem molha a cama.
As comidas mexicanas estavam lá apenas para
baixar sua guarda na esperança de fazer você
responder mais honestamente. Obrigado novamente.

true
```

Um programa lindo e longo, com um monte de repetição. (Todas as seções de código que giram em torno de questões sobre comida mexicana são idênticas, e a questão sobre xixi na cama é ligeiramente diferente.) Repetição é uma coisa ruim. Mas nós não podemos fazer um grande iterador, porque algumas vezes nós queremos fazer alguma coisa entre as questões. Em situações como essa, é melhor escrever um método. Veja como:

```
def digaMoo
  puts 'mooooooo...'
end
```

```

```

Ahn... Nosso programa não disse **`mooooooo...`**. Por que não? Porque nós não o mandamos fazer isso. Nós apenas dissemos *como* fazer para dizer **`mooooooo...`**, mas nós nunca o mandamos *fazer* isso. Vamos lá, outra tentativa:

```
def digaMoo
  puts 'mooooooo...'
end

digaMoo
digaMoo
puts 'coin-coin'
digaMoo
digaMoo
```

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

Ah! Muito melhor. (Para o caso de você não falar francês, havia um pato francês no meio do programa. Na França, os patos fazem *"coin-coin"*).

Então, nós **`def`**inimos o método **`digaMoo`** (Nomes de método, assim como nomes de variáveis, começam com uma letra minúscula. Há exceções, como **`+`** ou **`==`**). Mas métodos não têm de sempre estar associados com objetos? Bem, sim, eles têm. E nesse caso (assim como com o **`puts`** e o **`gets`**), o método está associado apenas com o objeto representando o programa como um todo. No próximo capítulo nós vamos ver como adicionar métodos a outros objetos. Mas primeiro...

Parâmetros de Métodos

Você deve ter notado que se podemos chamar alguns métodos (como o **`gets`**, ou o **`to_s`**, ou o **`reverse`**...) apenas como um objeto. Porém, outros métodos (como o **`+`**, o **`-`**, o **`puts`**...) recebem *parâmetros* para dizer ao objeto o que fazer com o método. Por exemplo, você não diz apenas **`5+`**, certo? Você está dizendo ao **`5`** para adicionar, mas você não está dizendo *o que* adicionar.

Para adicionar um parâmetro ao **`digaMoo`** (o número de mugidos, por exemplo), nós podemos fazer o seguinte:

```
def digaMoo numeroDeMoos
  puts 'mooooooo...'*numeroDeMoos
end

digaMoo 3
puts 'oink-oink'
digaMoo  #  Isso vai dar erro porque não foi passado nenhum parâmetro.
```

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

numeroDeMoos é uma variável que aponta para o parâmetro que foi passado. Vou dizer mais uma vez, mas é um pouco confuso: **numeroDeMoos** é uma variável que aponta para o parâmetro que foi passado. Então, se eu digitar **digaMoo 3**, o parâmetro é o **3**, e a variável **numeroDeMoos** aponta para **3**.

Como você pode ver, agora o parâmetro é *necessário*. Afinal, o que o **digaMoo** deve fazer é multiplicar **'mooooooo'** por um número. Mas por quanto, se você não disse? Seu computador não tem a menor idéia.

Se compararmos os objetos em Ruby aos substantivos em português, os métodos podem, da mesma forma, ser comparados aos verbos. Assim, você pode imaginar os parâmetros como advérbios (assim como em **digaMoo**, onde o parâmetro nos diz *como* a **digaMoo** agir) ou algumas vezes como objetos diretos (como em **puts**, onde o parâmetro é *o que* o **puts** irá imprimir).

Variáveis Locais

No programa a seguir, há duas variáveis:

```
def dobreEste num
  numVezes2 = num*2
  puts 'O dobro de '+num.to_s+' é '+numVezes2.to_s
end

dobreEste 44
```

```
O dobro de 44 é 88
```

As variáveis são **num** e **numVezes2**. Ambas estão localizadas dentro do método **dobreEste**. Estas (e todas as outras variáveis que você viu até agora) são *variáveis locais*. Isso significa que elas vivem dentro do método e não podem sair. Se você tentar, você terá um erro:

```
def dobreEste num
  numVezes2 = num*2
  puts 'O dobro de '+num.to_s+' é '+numVezes2.to_s
end

dobreEste 44
puts numVezes2.to_s
```

```
O dobro de 44 é 88
#<NameError: undefined local variable or method `numVezes2' for #<StringIO:0x7f237d5e063
```

Variável local não definida... Na verdade, nós *definimos* aquela variável local, mas ela não é local em relação ao local onde tentamos usá-la; ela é local ao método.

Isso pode parecer inconveniente, mas é muito bom. Enquanto você não tiver acesso a variáveis de dentro dos métodos, isso também quer dizer que ninguém tem acesso às *suas* variáveis, e isso quer dizer que ninguém pode fazer algo como isso:

```
def pequenaPeste var
  var = nil
  puts 'HAHA! Eu acabei com a sua variável!'
end

var = 'Você não pode tocar na minha variável!'
pequenaPeste var
puts var
```

```
HAHA! Eu acabei com a sua variável!
Você não pode tocar na minha variável!
```

Há, atualmente, *duas* variáveis naquele pequeno programa chamadas **var**: uma dentro do método **pequenaPeste** e uma fora dele. Quando você chamou **pequenaPeste var**, nós realmente passamos a string que estava em **var** para a outra, então as mesmas estavam apontando para a mesma string. Então, o método **pequenaPeste** apontou a sua **var** *local* para **nil**, mas isso não fez nada com a **var** de fora do método.

Retornando Valores

Você deve ter notado que alguns métodos devolvem alguma coisa quando você os chama. Por exemplo, o método **gets** *retorna* uma string (a string que você digitou), e o método **+** em **5+3**, (que é, na verdade **5.+(3)**) retorna **8**. Os métodos aritméticos para números retornam números, e os métodos aritméticos para strings retornam strings.

É importante entender a diferença entre métodos retornando um valor para onde ele foi chamado, e o seu programa gerando uma saída para a sua tela, como o **puts** faz. Note que **5+3** retorna **8**; ele **não** imprime **8** na tela.

Então, *o que* o **puts** retorna? Nós nunca nos importamos antes, mas vamos dar uma olhadinha agora:

```
valorRetorno = puts 'Este puts retornou:'
puts valorRetorno
```

```
Este puts retornou:
nil
```

O primeiro **puts** retornou **nil**. Apesar de não termos testado o segundo **puts**, ele fez a mesma coisa; **puts** sempre retorna um **nil**. Todo método tem que retornar alguma coisa, mesmo que seja apenas um **nil**.

Faça uma pausa e escreva um programa que encontre o que o método **digaMoo** retornou.

Você está surpreso? Bem, as coisas funcionam assim: o valor de retorno de um método é simplesmente a última linha avaliada do método. No caso do método **digaMoo**, isso quer dizer que ele retornou **'puts mooooooo...'*numeroDeMoos**, que é um simples **nil**, já que **puts** sempre retorna um **nil**. Se nós quisermos que todos os nossos métodos retornem a string **'yellow submarine'**, nós apenas temos que colocar ela no fim deles:

```
def digaMoo numeroDeMoos
  puts 'mooooooo...'*numeroDeMoos
```

```
  'yellow submarine'
end

x = digaMoo 2
puts x
```

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

Agora vamos tentar aquela pesquisa de psicologia de novo, mas desta vez vamos escrever um método que faça a pergunta para nós. Ele vai precisar pegar a questão como um parâmetro e retornar **`true`** se a resposta foi `sim` e **`false`** se a resposta foi `não` (Mesmo que nós ignoremos a resposta na maioria das vezes, é uma boa idéia fazer o método retornar a resposta. Assim nós podemos usar isso para a questão sobre molhar a cama). Eu também vou dar uma resumida na saudação e no final, apenas para ficar mais fácil de ler:

```
def pergunte pergunta
  boaResposta = false
  while (not boaResposta)
    puts pergunta
    replica = gets.chomp.downcase

    if (replica == 'sim' or replica == 'não')
      boaResposta = true
      if replica == 'sim'
        resposta = true
      else
        resposta = false
      end
    else
      puts 'Por favor, responda com "sim" ou "não".'
    end
  end

  resposta #  É isso o que será retornado (true ou false).
end

puts 'Olá e obrigado por...'
puts

pergunte 'Você gosta de comer tacos?'          #  Nós ignoramos o valor de retorno desse
pergunte 'Você gosta de comer burritos?'
molhaCama = pergunte 'Você faz xixi na cama?'  #  Nós salvamos o retorno desse.
pergunte 'Você gosta de comer chimichangas?'
pergunte 'Você gosta de comer sopapillas?'
pergunte 'Você gosta de comer tamales?'
puts 'Apenas mais algumas perguntas...'
pergunte 'Você gosta de beber horchata?'
pergunte 'Você gosta de comer flautas?'

puts
puts 'QUESTIONÁRIO:'
puts 'Obrigado por...'
puts
puts molhaCama
```

```
Olá e obrigado por...

Você gosta de comer tacos?
sim
Você gosta de comer burritos?
sim
Você faz xixi na cama?
de jeito nenhum!
Por favor, responda com "sim" ou "não".
Você faz xixi na cama?
NÃO
```

```
Por favor, responda com "sim" ou "não".
Você faz xixi na cama?
sim
Você gosta de comer chimichangas?
sim
Você gosta de comer sopapillas?
sim
Você gosta de comer tamales?
sim
Apenas mais algumas perguntas...
Você gosta de beber horchata?
sim
Você gosta de comer flautas?
sim

QUESTIONÁRIO:
Obrigado por...

true
```

Nada mal, hein? Nós podemos adicionar mais perguntas (e adicionar mais perguntas é *fácil* agora), mas nosso programa continuará pequeno! Isso é um grande progresso — o sonho de todo programador preguiçoso.

Mais um Grande Exemplo

Eu acho que outro exemplo de método seria muito útil aqui. Vamos chamar esse de **`numeroPortugues`**. Esse método vai pegar um número, como o **`22`**, e retorná-lo por extenso (nesse caso, a string **`vinte e dois`**). Por enquanto, vamos fazê-lo trabalhar apenas com inteiros entre **`0`** e **`100`**.

*(**NOTA:** Esse método usa um novo truque para retornar a partir de um método antes do fim usando a palavra-chave **`return`**, e introduz um novo conceito: **`elsif`**. Isso deve ficar mais claro no contexto, mostrando como as coisas funcionam).*

```
def numeroPortugues numero
  #  Nós apenas queremos números entre 0 e 100.
  if numero < 0
    return 'Por favor, entre com um número maior ou igual a zero.'
  end
  if numero > 100
    return 'Por favor, entre com um número menor ou igual a 100.'
  end

  numExtenso = ''  #  Esta é a string que vamos retornar.

  #  "falta" é quanto do número ainda falta para escrevermos.
  #  "escrevendo" é a parte que estamos escrevendo agora.
  falta    = numero
  escrevendo = falta/100                #  Quantas centenas faltam escrever?
  falta    = falta - escrevendo*100     #  Subtraia essas centenas.

  if escrevendo > 0
    return 'cem'
  end

  escrevendo = falta/10               #  Quantas dezenas faltam escrever?
  falta  = falta - escrevendo*10      #  Subtraia essas dezenas.

  if escrevendo > 0
    if escrevendo == 1  #  Oh-oh...
      #  Já que não podemos escrever "dez e dois",
      #  vamos fazer algo especial aqui
      if falta == 0
        numExtenso = numExtenso + 'dez'
      elsif falta == 1
        numExtenso = numExtenso + 'onze'
      elsif falta == 2
```

```
      numExtenso = numExtenso + 'doze'
    elsif falta == 3
      numExtenso = numExtenso + 'treze'
    elsif falta == 4
      numExtenso = numExtenso + 'catorze'
    elsif falta == 5
      numExtenso = numExtenso + 'quinze'
    elsif falta == 6
      numExtenso = numExtenso + 'dezesseis'
    elsif falta == 7
      numExtenso = numExtenso + 'dezessete'
    elsif falta == 8
      numExtenso = numExtenso + 'dezoito'
    elsif falta == 9
      numExtenso = numExtenso + 'dezenove'
    end
    #  Já que já cuidamos das unidades,
    #  não temos mais nada a escrever.
    falta = 0
  elsif escrevendo == 2
    numExtenso = numExtenso + 'vinte'
  elsif escrevendo == 3
    numExtenso = numExtenso + 'trinta'
  elsif escrevendo == 4
    numExtenso = numExtenso + 'quarenta'
  elsif escrevendo == 5
    numExtenso = numExtenso + 'cinqüenta'
  elsif escrevendo == 6
    numExtenso = numExtenso + 'sessenta'
  elsif escrevendo == 7
    numExtenso = numExtenso + 'setenta'
  elsif escrevendo == 8
    numExtenso = numExtenso + 'oitenta'
  elsif escrevendo == 9
    numExtenso = numExtenso + 'noventa'
  end

  if falta > 0
    numExtenso = numExtenso + ' e '
  end
end

escrevendo = falta  #  Quantos ainda faltam a escrever?
falta   = 0         #  Subtraia esses.

if escrevendo > 0
  if escrevendo == 1
    numExtenso = numExtenso + 'um'
  elsif escrevendo == 2
    numExtenso = numExtenso + 'dois'
  elsif escrevendo == 3
    numExtenso = numExtenso + 'três'
  elsif escrevendo == 4
    numExtenso = numExtenso + 'quatro'
  elsif escrevendo == 5
    numExtenso = numExtenso + 'cinco'
  elsif escrevendo == 6
    numExtenso = numExtenso + 'seis'
  elsif escrevendo == 7
    numExtenso = numExtenso + 'sete'
  elsif escrevendo == 8
    numExtenso = numExtenso + 'oito'
  elsif escrevendo == 9
    numExtenso = numExtenso + 'nove'
  end
end

if numExtenso == ''
  #  A única forma de "numExtenso" estar vazia é
  #  se o "numero" for 0
  return 'zero'
end
```

```
  #  Se chemagmos aqui, então temos um
  #  número entre 0 e 100, então precisamos
  #  apenas retornar o "numExtenso"
  numExtenso
end

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

```
zero
nove
dez
onze
dezessete
trinta e dois
oitenta e oito
noventa e nove
cem
```

Bem, ainda há algumas coisas nesse programa que eu não gostei. Primeiro: há muita repetição. Segundo: esse programa não consegue lidar com números maiores do que 100. Terceiro: há muitos casos especiais, muitos retornos (**return**). Vamos usar alguns vetores e tentar dar uma limpada:

```
def numeroPortugues numero
  if numero < 0  #  Nada de números negativos.
    return 'Por favor, digite um número positivo.'
  end
  if numero == 0
    return 'zero'
  end

  #  Nada de casos especiais! Nada de retornos!

  numExtenso = ''  #  Esta é a string que vamos retornar.

  unidades     = ['um'  , 'dois', 'tres', 'quatro', 'cinco',
                  'seis', 'sete', 'oito', 'nove']
  dezenas      = ['dez'     ,   'vinte'   ,    'trinta' ,   'quarenta',    'cinqüenta',
                  'sessenta',   'sessenta',    'oitenta',   'noventa']
  especiais =    ['onze'     ,  'doze'    ,    'treze'  , 'catorze', 'quinze',
                  'dezesseis',  'dezesete',    'dezoito', 'dezenove']

  #  "falta" é quanto do número ainda falta escrever.
  #  "escrevendo" é a parte que estamos escrevendo agora.
  falta  = numero
  escrevendo = falta/100           #  Quantas centenas ainda faltam escrever?
  falta  = falta - escrevendo*100  #  Subtraia essas centenas.

  if escrevendo > 0
    #  Aqui está o truque sujo:
    centenas   = numeroPortugues escrevendo
    numExtenso = numExtenso + centenas + ' centos'
    #  Isso é chamado "recursão". O que nós fizemos?
    #  Eu disse para o método chamar a si mesmo, mas
    #  passando "escrevendo" como parâmetro, ao invés
    #  de "numero". Lembre-se de que "escrevendo" é
    #  (agora) o número de dezenas que nós estamos escrevendo.
    #  Depois de adicionarmos as "centenas" a "numExtenso",
    #  nós adicionamos a string " centos". Então, se nós
    #  chamamos numeroPortugues com 1999 (numero = 1999),
    #  agora escrevendo será 19, e "falta" deve ser 99.
```

```
    #  A coisa mais preguiçosa que fazemos aqui é
    #  mandar o método numeroPortugues escrever o número
    #  19 por extenso, e então adicionando "centos" ao
    #  fim e escrevendo "noventa e nove" ao que falta.
    #  Ficando, portanto, "dezenove centos e noventa e nove".

    if falta > 0
      #  Nós não escrevemos dois centosecinqüenta e um'...
      numExtenso = numExtenso + ' e '
    end
  end

  escrevendo = falta/10           #  Quantas dezenas faltam escrever?
  falta  = falta - escrevendo*10  #  Subtraia dessas dezenas.

  if escrevendo > 0
    if ((escrevendo == 1) and (falta > 0))
      #  Não podemos escrever "dez e dois", temos que escrever "doze",
      #  então vamos fazer uma exceção.
      numExtenso = numExtenso + especiais[falta-1]
      #  O "-1" aqui é porque especiais[3] é 'catorze', e não 'treze'.

      #  Já que cuidamos do dígito das unidades,
      #  não falta mais nada
      falta = 0
    else
      numExtenso = numExtenso + dezenas[escrevendo-1]
      #  E o "-1" aqui é porque dezenas[3] é 'quarenta', e não 'trinta'.
    end

    if falta > 0
      #  Como nós não escrevemos "sessentaequatro"...
      numExtenso = numExtenso + ' e '
    end
  end

  escrevendo = falta  #  Quantas unidades faltam ser escritas?
  falta  = 0          #  Subtraia elas.

  if escrevendo > 0
    numExtenso = numExtenso + unidades[escrevendo-1]
    #  Novamente: O "-1" aqui é porque unidades[3] é 'quatro', e não 'três'.
  end

  #  Agora podemos, simplesmente, retornar o nosso "numExtenso"...
  numExtenso
end

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

```
zero
nove
dez
onze
dezesete
trinta e dois
oitenta e oito
noventa e nove
```

```
um centos
um centos e um
dois centos e trinta e quatro
trinta e dois centos e onze
noventa e nove centos e noventa e nove centos e noventa e nove
um centos centos centos centos centos centos
```

Ahhhh.... Agora está muito melhor. O programa está um pouco maçante, mas é porque eu enchi de comentários. Agora ele funciona para números grandes... embora não de uma maneira muito elegante. Por exemplo, eu acho que **`'um trilhão'`** seria muito mais elegante para o último número, ou mesmo **`'um milhão milhão'`** (muito embora todas as três estejam corretas). Na verdade, você pode fazer isso agora...

Algumas Coisinhas para Tentar

• Melhore o método **`numeroPortugues`**. Primeiro, coloque os milhares. Ou seja, ele deve retornar **`'um mil'`** ao inves de **`'dez centos'`** e **`'dez mil'`** ao invés de **`'um centos centos'`**. Seria interessante retornar **`'cem'`**, **`'duzentos'`** e etc. ao invés de **`'um centos'`**, **`'dois centos'`** e etc. (muito embora ambas as formas estejam corretas).

• Melhore o método **`numeroPortugues`** um pouquinho mais. Coloque milhões agora, assim você conseguirá coisas como **`'um milhão'`** ao invés de **`'um mil mil'`**. Depois tente adicionar bilhões e trilhões. Quão longe você consegue ir?

• *"Um elefante incomoda muita gente..."* Usando o método **`numeroPortugues`**, escreva esse clássico *corretamente* agora. Coloque seu computador de castigo: faça ele contar 9999 elefantes (Não exagere nos elefantes, porém. Escrever todos esses elefantes na sua tela vai demorar um pouco. Se você pensar em um milhão de elefantes, você vai acabar se colocando de castigo!).

Parabéns! Agora você já é um verdadeiro programador! Você aprendeu tudo o que você precisava para escrever grandes programas do zero. Se você tiver idéias de programas que você gostaria de escrever para você mesmo, tente agora!

Claro que escrever tudo do zero pode ser um processo muito lento. Por que gastar tempo e energia escrevendo um código que alguém já escreveu? Você quer um programa que mande alguns e-mails? Você gostaria de salvar e carregar arquivos em seu computador? Que tal gerar páginas da web para um tutorial onde os exemplos de código sejam executados cada vez que a página é carregada? :) Ruby tem muitos tipos diferentes de objetos que podemos usar para nos ajudar a escrever programas mais rapidamente.

Aprenda a Programar

9. Classes

Até agora, nós vimos muitos tipos diferentes de objetos, ou *classes*: strings, inteiros, ponto flutuante, vetores e alguns objetos especiais (**`true`**, **`false`** e **`nil`**), que vamos voltar a falar mais tarde. Em Ruby, todas essas classes sempre começam com maiúsculas: **`String`**, **`Integer`** (Inteiros), **`Float`** (Ponto Flutuante), **`Array`** (Vetores) e etc. Geralmente, se queremos criar um novo objeto de uma certa classe, nós usamos o **`new`**:

```
a = Array.new  + [12345]  #  Adição de Vetores.
b = String.new + 'olá'  #  Adição com Strings.
c = Time.new

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

```
a = 12345
b = olá
c = Tue Jun 12 15:42:19 -0700 2012
```

Como nós podemos criar vetores e strings usando **`[...]`** e **`'...'`**, respectivamente, nós raramente usamos o **`new`** para isso (De qualquer forma, não está muito claro, no exemplo anterior, que **`String.new`** cria uma string vazia e que **`Array.new`** cria um vetor vazio). Números, porém, são uma exceção: você não pode criar um inteiro usando **`Integer.new`**. Você apenas tem que digitar o número.

A classe Time

Está bem, e a classe **`Time`**? Objetos **`Time`** representam momentos de tempo. Você pode adicionar (ou subtrair) números para (ou de) tempos para conseguir novos instantes: adicionando **`1.5`** a um instante, retorna um novo instante de um segundo e meio depois:

```
tempo  = Time.new    #  O instante em que você carrega esta página.
tempo2 = tempo + 60  #  Um minuto depois.

puts tempo
puts tempo2
```

```
Tue Jun 12 15:42:19 -0700 2012
Tue Jun 12 15:43:19 -0700 2012
```

Você pode, também, fazer um tempo para um momento específico usando **`Time.mktime`**:

```
puts Time.mktime(2000, 1, 1)          #  Ano 2000.
puts Time.mktime(1976, 8, 3, 10, 11)  #  Ano em que nasci.
```

```
Sat Jan 01 00:00:00 -0800 2000
Tue Aug 03 10:11:00 -0700 1976
```

Nota: quando eu nasci, estava em uso o Horário de Verão do Pacífico (PDT, em Inglês). Quanto o ano 2000 chegou, porém, estava em uso o Horário Padrão do Pacífico (PST, em Inglês), pelo menos para nós, da costa Oeste. Os parênteses servem para agrupar os parâmetros para o **`mktime`**. Quanto mais parâmetros você adicionar, mais preciso o seu instante se tornará.

Você pode comparar dois tempos utilizando os métodos de comparação (um tempo anterior é *menor que*

Algumas Coisinhas Para Tentar

• Um bilhão de segundos... Encontre o segundo exato do seu nascimento (se você puder). Descubra quando você fará (ou quando você fez?) um bilhão de segundos de idade. Então vá marcar na sua folhinha.

• Feliz Aniversário! Pergunte o ano de nascimento em que uma pessoa nasceu. Então pergunte o mês e, finalmente, o dia. Então descubra a idade dessa pessoa e lhe dê um **PUXÃO DE ORELHA!** para cada aniversário que ela fez.

A Classe Hash

Outra classe muito útil é a classe **Hash**. Hashes são muito parecidos com vetores: eles têm um monte de espaços que podem conter vários objetos. Porém, em um vetor, os espaços são dispostos em uma linha, e cada um é numerado (iniciando pelo zero). Em um Hash, porém, os espaços não estão dispostos em uma linha (eles estão apenas juntos), e você pode usar *qualquer* objeto para se referir a um espaço, não apenas um número. É bom usar hashes quando você tem uma porção de coisas que você quer armazenar, mas que não têm, realmente, uma ordem. Por exemplo, as cores que eu uso em diversas partes desse tutorial:

```
colorArray = []  #  o mesmo que Array.new
colorHash  = {}  #  o mesmo que Hash.new

colorArray[0]          = 'vermelho'
colorArray[1]          = 'verde'
colorArray[2]          = 'azul'
colorHash['strings']   = 'vermelho'
colorHash['numbers']   = 'verde'
colorHash['keywords'] = 'azul'

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

```
vermelho
verde
azul
strings:  vermelho
keywords:  azul
numbers:  verde
```

Se eu usar um vetor, eu tenho que me lembrar que o espaço **0** é para strings, o slot **1** é para números e etc. Mas se eu usar um Hash, fica fácil! O espaço **'strings'** armazena a cor das strings, claro. Nada para lembrar. Você deve ter notado que quando eu usei o **each**, os objetos no hash não vieram na mesma ordem que eu os coloquei (Pelo menos não quando eu escrevi isso. Talvez agora esteja em ordem... você nunca sabe a ordem com os hashes). Vetores servem para colocar as coisas em ordem, os Hashes não.

Apesar das pessosas normalmente usarem strings para nomear os espaços em um hash, você pode usar qualquer tipo de objeto, até mesmo vetores e outros hashes (apesar de eu não conseguir achar uma razão para você fazer isso...):

```
hashBizarro = Hash.new

hashBizarro[12] = 'macacos'
hashBizarro[[]] = 'totalmente vazio'
hashBizarro[Time.new] = 'nada melhor que o Presente'
```

Hashes e vetores são bons para coisas diferentes: a escolha sobre qual resolve o seu problema melhor é sua, e diferente para todos os problemas que você tiver.

Expandindo Classes

No fim do último capítulo, você escreveu um método para retornar um número por extenso. Porém, esse não era um método de inteiros: era um método genérico do programa. Não seria mais legal se você pudesse escrever **22.ext** ao invés de **porExtenso 22**? Olha só como você pode fazer isso:

```
class Integer

    def ext
    if self == 5
      porExtenso = 'cinco'
    else
      porExtenso = 'cinqüenta e oito'
    end

    porExtenso
  end

end

#  Eu prefiro testar sempre em duplas...
puts 5.ext
puts 58.ext
```

```
cinco
cinqüenta e oito
```

Bem, eu testei; e nada explodiu. :)

Nós definimos um método inteiro apenas "pulando" dentro da classe **`Integer`**, definindo o método lá dentro e caindo fora. Agora todos os inteiros tem esse sensacional (incompleto) método. Na verdade, se você não gostar da forma como o método nativo **`to_s`** faz as coisas, você pode simplesmente redefini-lo da mesma forma... mas eu não recomendo isso! É melhor deixar os métodos antigos quietos em seu canto e fazer novos quando você precisar de uma coisa nova.

Confuso ainda? Deixe-me voltar até o último programa mais um pouco. Até agora, quando nós executamos qualquer código ou definido um método, nós o fizemos no objeto "programa" padrão. No nosso último programa, nós saímos daquele objeto pela primeira vez e fomos para dentro da classe **`Integer`**. Nós definimos um método lá (o que o tornou um método inteiro) e todos os inteiros podem usar ele. Dentro daquele métodos, nós usamos o **`self`** para nos referir ao objeto (o inteiro) que estiver usando o método.

Criando Classes

Nós já vimos um monte de objetos de classes diferentes. Porém, é fácil criar tipos de objeto que o Ruby não tenha. Por sorte, criar uma classe nova é tão fácil como expandir uma classe já existente. Vamos supor que eu queira rodar alguns dados no Ruby. Olhe como podemos fazer uma classe chamada Dado:

```
class Dado

  def rolar
    1 + rand(6)
  end

end

#  Vamos fazer dois dados...
dados = [Dado.new, Dado.new]

#  ...e rolar cada um deles.
dados.each do |dado|
  puts dado.rolar
end
```

```
1
4
```

(Se você pulou a parte que falava sobre números aleatórios, **`rand(6)`** apenas devolve um número aleatório entre **`0`** e **`5`**).

Só isso! Objetos de nossa própria autoria. Role os dados algumas vezes (utilizando o botão de "Atualizar" do seu navegador) e veja o que acontece.

Nós podemos definir todo o tipo de métodos para os nossos objetos... mas tem alguma coisa errada. Trabalhando com esses objetos não mudou grande coisa desde que aprendemos a mexer com variáveis. Olhe o nosso dado, por exemplo. Cada vez que rolamos ele, nós temos um número diferente. Mas se nós quisermos salvar aquele número, nós temos que criar uma variável e apontar para aquele número. E qualquer dado que preste deve *ter* um número, e rolando o dado deve mudar o número. Se nós armazenarmos o dado, nós não temos como saber qual número ele está mostrando.

Porém, se nós tentarmos armazenar o número que nós tirmaos no dado em uma variável (local) dentro de **rolar**, o valor será perdido assim que o **rolar** acabar. Nós precisamos salvar esse número em um tipo diferente de variável:

VARIÁVEIS DE INSTÂNCIA

Normalmente quando falamos sobre strings, nós apenas nos referimos a elas como *strings*. Porém, nós poderíamos chamá-las de *Objetos do tipo String*. Algumas vezes, alguns programadores podem chamá-las de *instâncias da classe* ***String***, mas essa é uma forma exagerada (e muito longa) de dizer *string*. Uma *instância* de uma classe é apenas um objeto daquela classe.

Portanto, variáveis de instância são como variáveis de objeto. Uma variável local de um método ficam vivas até que o método termine. Uma variável de instância de um objeto, por outro lado, ficará viva enquanto o objeto estiver vivo. Para diferenciar variáveis de instância de variáveis locais, elas têm uma `@` na frente dos seus nomes:

```
class Dado

  def rolar
    @numeroMostrado = 1 + rand(6)
  end

  def mostrado
    @numeroMostrado
  end

end

dado = Dado.new
dado.rolar
puts dado.mostrado
puts dado.mostrado
dado.rolar
puts dado.mostrado
puts dado.mostrado
```

```
4
4
2
2
```

Muito legal! Agora o **rolar** rola o dado e o **mostrado** nos diz qual é o número que saiu. Mas e se quisermos ver qual número saiu antes de rolar o dado (antes de termos definido **@numeroMostrado**)?

```
class Dado

  def rolar
    @numeroMostrado = 1 + rand(6)
  end

  def mostrado
    @numeroMostrado
  end

end

#  Já que eu não vou mais usar esse dado,
#  eu não preciso salvá-lo em uma variável.
puts Dado.new.mostrado
```

```
nil
```

Hum... Bem, pelo menos não deu erro. Espera aí, não faz muito sentido um dado "não-rolado" ou o que quer que **nil** signifique aqui. Seria muito mais bacana se nós pudessemos rolar o dado assim que ele for criado. É isso que o **initialize** faz:

```
class Dado

  def initialize
    #  Eu vou apenas rolar o dado, apesar de
    #  podermos fazer qualquer coisa que
    #  queiramos fazer, como colocar a face '6'
    #  para cima
```

```
    rolar
  end

  def rolar
    @numeroMostrado = 1 + rand(6)
  end

  def mostrado
    @numeroMostrado
  end

end

puts Dado.new.mostrado
```

```
1
```

Quando um objeto é criado, o método **`initialize`** (se foi definido) é sempre chamado.

Nosso dado está quase perfeito. A única coisa que falta é uma maneira de arrumar qual número está sendo mostrado... Por que você não escreve o método **`trapaca`** que faça isso? Volte quando tiver terminado (e quando você testar e funcionar, lógico). Apenas tenha certeza de que ninguém pode fazer com o que o dado mostre um **7**!

Foi muito legal o que fizemos até agora. Mas foi apenas uma brincadeira, mesmo assim. Deixe-me mostrar um exemplo mais interessante. Vamos fazer um bichinho virtual, um dragão bebê. Assim como todos os bebês, ele deve conseguir comer, dormir e "atender à natureza", o que significa que vamos ter que ter como alimentá-lo, colocá-lo pra dormir e levar ele até o quintal. Internamente, o nosso dragão precisa saber se está com fome, cansado ou se precisa ir lá fora, mas nós não poderemos ver isso enquanto estivermos interagindo com ele, assim como você não pode perguntar a um bebê "você está com fome?". Então nós vamos adicionar algumas maneiras legais para interagir com nosso dragão bebê, e quando ele nascer nós vamos dar um nome para ele (Qualquer coisa que você passe como parâmetro para o método **`new`** será passado para o método **`initialize`** para você). Certo, vamos tentar:

```
class Dragao

  def initialize nome
    @nome = nome
    @dormindo = false
    @comidaEstomago  = 10 #  Ele está cheio
    @comidaIntestino =  0 #  Ele não precisa ir ao quintal

    puts @nome + ' nasceu.'
  end

  def alimentar
    puts 'Você alimentou o ' + @nome + '.'
    @comidaEstomago = 10
    passagemDeTempo
  end

  def quintal
    puts 'Você levou o ' + @nome + ' até o quintal.'
    @comidaIntestino = 0
    passagemDeTempo
  end

  def colocarNaCama
    puts 'Você colocou o ' + @nome + ' na cama.'
    @dormindo = true
    3.times do
      if @dormindo
        passagemDeTempo
      end
      if @dormindo
        puts @nome + ' está roncando e enchendo o quarto de fumaça.'
      end
    end
    if @dormindo
      @dormindo = false
      puts @nome + ' está acordando.'
    end
  end

  def jogar
    puts 'Você joga o ' + @nome + ' no ar.'
    puts 'Ele dá uma risadinha e queima suas sobrancelhas.'
    passagemDeTempo
  end
```

```
  def balancar
    puts 'Você balança o ' + @nome + ' gentilmente.'
    @dormindo = true
    puts 'Ele começa a cochilar...'
    passagemDeTempo
    if @dormindo
      @dormindo = false
      puts '...mas acorda quando você pára.'
    end
  end

  private

  #  "private" significa que os métodos definidos aqui
  #  são métodos internos do objeto. (Você pode
  #  alimentá-lo, mas você não pode perguntar se
  #  ele está com fome.)

  def comFome?
    #  Nomes de métodos podem acabar com "?".
    #  Normalmente, nós fazemos isso apenas
    #  se o métodos retorna verdadeiro ou falso,
    #  como esse:
    @comidaEstomago <= 2
  end

  def precisaSair?
    @comidaIntestino >= 8
  end

  def passagemDeTempo
    if @comidaEstomago > 0
      #  Mover a comida do estômago para o intestino.
      @comidaEstomago  = @comidaEstomago  - 1
      @comidaIntestino = @comidaIntestino + 1
    else  #  Nosso dragão está faminto!
      if @dormindo
        @dormindo = false
        puts 'Ele está acordando!'
      end
      puts @nome + ' está faminto! Em desespero, ele comeu VOCÊ!'
      exit  #  Isso sai do programa.
    end

    if @comidaIntestino >= 10
      @comidaIntestino = 0
      puts 'Ops!  ' + @nome + ' teve um acidente...'
    end

    if comFome?
      if @dormindo
        @dormindo = false
        puts 'Ele está acordando!'
      end
      puts 'O estômago do ' + @nome + ' está roncando...'
    end

    if precisaSair?
      if @dormindo
        @dormindo = false
        puts 'Ele está acordando!'
      end
      puts @nome + ' faz a dança para ir ao quintal...'
    end
  end

end

bichinho = Dragao.new 'Norbert'
bichinho.alimentar
bichinho.jogar
bichinho.quintal
bichinho.colocarNaCama
bichinho.balancar
bichinho.colocarNaCama
bichinho.colocarNaCama
bichinho.colocarNaCama
bichinho.colocarNaCama
```

UAU! Claro que seria muito mais legal se esse fosse um programa interativo, mas você pode fazer essa parte depois. Eu apenas

estava tentando mostrar as partes relacionadas diretamente a criar uma nova classe do tipo Dragão.

Nós dissemos um monte de coisas novas nesse exemplo. A primeira é simples: **`exit`** termina o programa onde estiver. A segunda é a palavra **`private`**, que nós colocamos bem no meio da nossa classe. Eu podia ter deixado ela de fora, mas eu apenas quis reforçar a idéia de que certos métodos você podia fazer com um dragão, enquanto que outros aconteciam com o dragão. Você pode pensar nisso como "coisas por trás dos panos": a não ser que você seja um mecânico de automóveis, tudo o que você precisa saber sobre carros é o acelerador, o freio e a direção. Um programador chama isso de *interface pública*

Agora, para um exemplo mais concreto nessa linha de raciocínio, vamos falar um pouco sobre como você representaria um carro em um jogo (o que é a minha linha de trabalho). Primeiro, você precisa decidir como irá se parecer sua interface pública; em outras palavras, quais métodos as pessoas podem chamar do seus objetos do tipo carro? Bem, eles devem podem acelerar e freiar, mas eles precisam, também, poder definir a força que estão aplicando no pedal (Há uma grande diferença entre tocar o acelerador e afundar o pé). Eles vão precisar também guiar, e novamente, e dizer que força estão aplicando na direção. Eu acho que você pode ir ainda mais longe e adicionar uma embreagem, piscas, lançador de foguetes, incinerador traseiro, um condensador de fluxo e etc... depende do tipo de jogo que você está fazendo.

Os objetos internos a um carro, porém, são mais complexos: outras coisas que um carro precisa são a velocidade, a direção e a posição (ficando no básico). Esses atributos serão modificados pressionando o pedal do acelerador ou o de freio e girando o volante, claro, mas o usuário não deve poder alterar essas informações diretamente (o que seria uma distorção). Você pode querer checar a derrapagem ou o dano, a resistência do ar e por aí vai. Tudo isso diz respeito apenas ao carro. Tudo isso é interno ao carro.

Algumas Coisinhas Para Tentar

• Faça uma classe de **`ArvoreDeLaranja`**. Ela deve ter um método **`altura`** que retorne sua altura, um método chamado **`passar_um_ano`** que, quando chamado, faz a árvore completar mais um ano de vida. Cada ano, a árvore cresce mais magra (não importa o quão grande você ache que uma árvore de laranja possa crescer em um ano), e depois de alguns anos (novamente, você faz as chamadas) a árvore deve morrer. Nos primeiros anos, ela não deve produzir frutos, mas depois de um tempo ela deve, e eu acho que as árvores mais velhas produzem muito mais frutos do que uma mais jovem com o passar dos anos... ou o que você achar mais lógico. E, é claro, você deve poder **`contar_as_laranjas`** (o número de laranjas na árvore), e **`pegar_uma_laranja`** (que irá reduzir o **`@numero_de_laranjas`** em um e retornar uma string dizendo quão deliciosa a laranja estava, ou então irá dizer que não há mais laranjas esse ano). Lembre-se de que as laranjas que você não pegar esse ano devem cair antes do próximo ano.

• Escreva um programa para que você possa interagir com o seu filhote de dragão. Você deve ser capaz de inserir comandos como `alimentar` e `quintal`, e esses métodos devem ser chamados no seu dragão. Logicamente que, como toda a entrada será por strings, você deve ter uma forma de *repassar os métodos*, onde seu programa deve validar a string digitada e chamar o método apropriado.

E isso é tudo! Mas espere um pouco... Eu não disse nada a você sobre classes para fazer coisas como mandar um e-mail, ou salvar e carregar arquivos do seu computador, ou como criar janelas e botões, ou mundos em 3D ou qualquer coisa! Bem, há apenas *muitas* classes que você pode usar, e isso torna impossível que eu mostre todas para você; mesmo eu não conheço todas elas. O que eu *posso* dizer para você é onde encontrar mais sobre elas, assim você pode aprender mais sobre as que você quiser usar. Mas antes de mandar você embora, há mais um recurso do Ruby que você deveria saber, algo que a maioria das outras linguagens não tem, mas que eu simplesmente não posso viver sem: blocos e procs.

Aprenda a Programar

10. Blocos e Procs

Este é, definitivamente, um dos recursos mais legais de Ruby. Algumas outras linguagens têm esse recurso, porém elas podem chamar isso de formas diferentes (como *closures*), mas muitas das mais populares não, o que é uma pena.

Então o que é essa nova coisa legal? É a habilidade de pegar um *bloco* de código (código entre **do** e **end**), amarrar tudo em um objeto (chamado de *proc*), armazenar isso em uma variável e passar isso para um método, e rodar o código do bloco quando você quiser (mais de uma vez, se você quiser). Então, é como se fosse um método, exceto pelo fato de que isso não está em um objeto (isso *é* um objeto), e você pode armazená-lo ou passá-lo adiante, como com qualquer outro objeto. Acho que é hora de um exemplo:

```
saudacao = Proc.new do
  puts 'Olá!'
end

saudacao.call
saudacao.call
saudacao.call
```

```
Olá!
Olá!
Olá!
```

Eu criei uma proc (eu acho que é abreviatura para "procedimento", mas o que importa é que rima com "block") que tem um bloco de código, então eu chamei (**call**) a proc três vezes. Como você pode ver, parece, em muito, com um método.

Atualmente, é muito mais parecido com um método do que eu mostrei para você, porque blocos podem receber parâmetros:

```
VoceGostade = Proc.new do |umaBoaCoisa|
  puts 'Eu *realmente* gosto de '+umaBoaCoisa+'!'
end

VoceGostade.call 'chocolate'
VoceGostade.call 'ruby'
```

```
Eu *realmente* gosto de chocolate!
Eu *realmente* gosto de ruby!
```

Certo, então nós vimos o que blocos e procs são e como os usar, mas e daí? Por que não usar apenas métodos? Bem, isso é porque existem algumas coisas que você não pode fazer com métodos. Particularmente, você não pode passar métodos para outros métodos (mas você pode passar procs para métodos), e métodos não podem retornar outros métodos (mas podem retornar procs). É apenas por isso que procs são objetos; métodos não.

(De qualquer forma, isso parece familiar? Sim, você já viu blocos antes... quando você aprendeu sobre iteradores. Mas vamos voltar a falar disso daqui a pouco.)

Métodos que Recebem Procs

Quando passamos uma proc em um método, nós podemos controlar como, se ou quantas vezes nós vamos chamar a proc. Por exemplo, posso dizer que há uma coisa que nós queremos fazer antes e depois que um código é executado:

```
def FacaUmaCoisaImportante umaProc
```

```
  puts 'Todo mundo apenas ESPERE! Eu tenho uma coisa a fazer...'
  umaProc.call
  puts 'Certo pessoal, Eu terminei. Voltem a fazer o que estavam fazendo.'
end

digaOla = Proc.new do
  puts 'olá'
end

digaTchau = Proc.new do
  puts 'tchau'
end

FacaUmaCoisaImportante digaOla
FacaUmaCoisaImportante digaTchau
```

```
Todo mundo apenas ESPERE! Eu tenho uma coisa a fazer...
olá
Certo pessoal, Eu terminei. Voltem a fazer o que estavam fazendo.
Todo mundo apenas ESPERE! Eu tenho uma coisa a fazer...
tchau
Certo pessoal, Eu terminei. Voltem a fazer o que estavam fazendo.
```

Talvez isso não pareça tão fabuloso... mas é. :-) É muito comum em programação que alguns requesitos críticos sejam executados. Se você grava um arquivo, por exemplo, você deve abrir o arquivo, escrever o que quiser lá dentro e então fechar o arquivo. Se você se esquecer de fechar o arquivo, Coisas Ruins(tm) podem acontecer. Mas toda a vez que você quiser salvar ou carregar um arquivo, você deve fazer a mesma coisa: abrir o arquivo, fazer o que você *realmente* quiser com ele e então fechar o arquivo. Isso é entediante e fácil de esquecer. Em Ruby, salvar (ou carregar) arquivos funciona similarmente com o código anterior, então você não precisa se preocupar com nada além de o que você quer salvar (ou carregar) (No próximo capítulo eu vou lhe mostrar como fazer coisas como salvar e carregar arquivos).

Você pode também escrever métodos que vão determinar quantas vezes, ou mesmo *se* uma proc será chamada. Aqui está um método que irá chamar uma proc metade do tempo, e outra que irá chamar a proc duas vezes.

```
def talvezFaca umaProc
  if rand(2) == 0
    umaProc.call
  end
end

def FacaDuasVezes umaProc
  umaProc.call
  umaProc.call
end

piscar = Proc.new do
  puts '<piscada>'
end

olhandofixamente = Proc.new do
  puts '<olhando fixamente>'
end

talvezFaca piscar
talvezFaca olhandofixamente
FacaDuasVezes piscar
FacaDuasVezes olhandofixamente
```

```
<piscada>
<piscada>
<olhando fixamente>
<olhando fixamente>
```

(Se você recarregar essa página algumas vezes, você verá que a saída muda) Esses são alguns dos usos mais comuns de procs, que nos possibilita fazer coisas que nós simplesmente não poderíamos fazer usando apenas métodos. Claro, você pode escrever um método que "pisque" duas vezes, mas você não pode escrever um que apenas faça *qualquer coisa* duas vezes!

Antes de seguirmos adiante, vamos olhar um último exemplo. Até agora, as procs que nós usamos foram muito similares umas às outras. Agora, elas serão um pouco diferentes, assim você poderá ver como um método depende das procs que lhe são passadas. Nosso método irá receber um objeto e uma proc e irá chamar essa proc naquele objeto. Se a proc retornar falso, nós saímos; caso contrário, nós chamamos a proc com o objeto retornado. Nós vamos continuar fazendo isso até que a proc retorne falso (o que é o melhor a fazer eventualmente, ou o programá irá travar). O método irá retornar o último valor não falso retornado pela proc.

```
def facaAteFalso primeiraEntrada, umaProc
  entrada = primeiraEntrada
  saida   = primeiraEntrada

  while saida
    entrada = saida
    saida   = umaProc.call entrada
  end

  entrada
end

construindoVetorDeQuadrados = Proc.new do |vetor|
  ultimoNumero = vetor.last
  if ultimoNumero <= 0
    false
  else
    vetor.pop                             # Jogue fora o último número...
    vetor.push ultimoNumero*ultimoNumero # ... e o substitua com esse quadrado...
    vetor.push ultimoNumero-1             # ... seguido pelo número imediatamente anterior.
  end
end

sempreFalso = Proc.new do |apenasIgnoreme|
  false
end

puts facaAteFalso([5], construindoVetorDeQuadrados).inspect
puts facaAteFalso('Estou escrevendo isso às 3:00; alguém me derrube!', sempreFalso)
```

```
[25, 16, 9, 4, 1, 0]
Estou escrevendo isso às 3:00; alguém me derrube!
```

Certo, esse foi um exemplo estranho, eu admito. Mas isso mostra como nosso método age diferentemente quando recebe diferentes procs.

O método **`inspect`** é muito parecido com o **`to_s`**, exceto pelo fato de que a string retornada tenta mostrar para você o código em ruby que está construindo o objeto que você passou. Aqui ele nos mostra todo o vetor retornado pela nossa primeira chamada a **`facaAteFalso`**. Você também deve ter notado que nós nunca elevamos aquele **`0`** ao quadrado, no fim do vetor. Já que **`0`** elevado ao quadrado continua apenas **`0`**, nós não precisamos fazer isso. E já que **`sempreFalso`** foi, você sabe, sempre **`falso`**, **`facaAteFalso`** não fez nada na segunda vez que a chamamos; apenas retornou o que lhe foi passada.

Métodos que Retornam Procs

Uma das outras coisas legais que você pode fazer com procs é criá-las em métodos e retorná-las. Isso permite toda uma variedade de poderes de programação malucos (coisas com nomes impressionantes, como *avaliação preguiçosa*, *estrutura de dados infinita*, e *temperando com curry*), mas o fato é de que eu nunca faço isso na prática e eu não me lembro de ter visto ninguém fazendo isso. Eu acho que isso é o tipo de coisa que você acaba não fazendo em Ruby, ou talvez Ruby apenas encoraje-o a achar outras soluções: eu não sei. De qualquer forma, eu vou tocar no assunto apenas brevemente.

Nesse exemplo, **`compor`** recebe duas procs e retorna uma nova proc que, quando chamada, chama uma terceira proc e passa seu resultado para a segunda proc.

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

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

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

dobreeEleve = compor dobre, quadrado
eleveeDobre = compor quadrado, dobre

puts dobreeEleve.call(5)
puts eleveeDobre.call(5)
```

```
100
50
```

Note que a chamada para **`proc1`** deve ser inserida entre parenteses dentro de **`proc2`**, para que seja executada primeiro.

Passando Blocos (e Não Procs) para Métodos

Certo, isso foi muito interessante academicamente, mas de pouca utilidade prática. Uma porção desse problema é que há três passos que você deve seguir (definir o método, construir a proc e chamar o método com a proc), quando eles podem ser resumidos em apenas dois (definir o método e passar o *bloco* diretamente ao método, sem usar uma proc), uma vez que na maior parte das vezes você não vai querer usar a proc ou o bloco depois que o passar para um método. Bem, você não sabe, mas Ruby tem isso para nós! Na verdade, você já estava fazendo isso todas as vezes que usou iteradores.

Eu vou mostrar a você um exemplo rápido, então nós vamos falar sobre isso.

```
class Array

    def cadaComparacao(&eraUmBloco_agoraUmaProc)
    eIgual = true  # Nós começamos com "verdadeiro" porque vetores começam com 0, mesmo se iguais.

    self.each do |objeto|
      if eIgual
        eraUmBloco_agoraUmaProc.call objeto
      end

      eIgual = (not eIgual)  # Comutando entre igual para diferente, ou de diferente para igual.
    end
  end

end

['maçã', 'maçã podre', 'cereja', 'mamona'].cadaComparacao do |fruta|
  puts 'Hum! Eu adoro tortas de '+fruta+', você não?'
end

#  Lembre-se, nós estamos pegando os mesmos elementos numerados
#  do array, todos que se relacionam com os outros números,
#  apenas porque gosto de causar esse tipo de problema.
[1, 2, 3, 4, 5].cadaComparacao do |bola_estranha|
  puts bola_estranha.to_s+' não é um número!'
end
```

```
Hum! Eu adoro tortas de maçã, você não?
Hum! Eu adoro tortas de cereja, você não?
1 não é um número!
3 não é um número!
5 não é um número!
```

Para passar um bloco para **`cadaComparacao`**, tudo o que temos que fazer é anexar o bloco após o método. Você pode passar um bloco para qualquer método dessa maneira, apesar de que muitos métodos vão apenas ignorar o bloco. Para fazer seu método *não* ignorar o bloco, mas pegá-lo e transformá-lo em uma proc, ponha o nome da proc no começo da lista dos parâmetros do seu método, precedido por um 'e' comercial (**`&`**). Essa parte é um pequeno truque, mas não é tão ruim, e você apenas precisa fazer isso uma vez (quando você define o método). Então você pode usar o método de novo, e de novo e de novo, assim como os métodos da linguagem que aceitam blocos, como o **`each`** e o **`times`** (Lembra-se do **`5.times do`**...?).

Se você estiver confuso, apenas lembre-se do que supostamente o método **`cadaComparacao`** faz: chama o bloco passado como parâmetro para cada elemento no vetor. Depois que você o escrever e ele estiver funcionando, você não vai precisar pensar sobre o que está acontecendo realmente por baixo dos panos ("qual bloco é chamado quando??"); na verdade, é exatamente *por isso* que escrevemos métodos assim: nós nunca mais vamos precisar pensar sobre como eles funcionam novamente. Nós vamos apenas usar-los.

Eu lembro que uma vez eu quis cronometrar quanto tempo cada seção do meu código estava demorando (Isso é algo conhecido como *sumarizar* o código). Então, eu escrevi um método que pegava o tempo antes de executar o código, o executava e então fazia uma nova medição do tempo e me retornava a diferença. Eu não estou conseguindo achar o código agora, mas eu não preciso disso: provavelmente é um código parecido com esse:

```
def sumario descricaoDoBloco, &bloco
  tempoInicial = Time.now
```

```
  bloco.call

  duracao = Time.now - tempoInicial

  puts descricaoDoBloco+': '+duracao.to_s+' segundos'
end

sumario 'dobrando 25000 vezes' do
  numero = 1

  25000.times do
    numero = numero + numero
  end

  puts numero.to_s.length.to_s+' dígitos'  #  É isso mesmo: o número de dígitos nesse número ENORME.
end

sumario 'contando até um milhão' do
  numero = 0

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

```
7526 dígitos
dobrando 25000 vezes: 0.087939 segundos
contando até um milhão: 0.164505 segundos
```

Que simplicidade! Que elegância! Com aquele pequeno método eu posso, agora, facilmente cronometrar qualquer seção, de qualquer programa, que eu queira, eu apenas preciso jogar o código para um bloco e enviar ele para o **`sumario`**. O que poderia ser mais simples? Em muitas linguagens, eu teria que adicionar explicitamente o código de cronometragem (tudo o que está em **`sumario`**) em volta de qualquer seção que eu queira medir. Em Ruby, porém, eu deixo tudo em um só lugar, e (o mais importante) fora do meu caminho!

Algumas Coisinhas Para Tentar

• *Relógio Cuco*. Escreva um método que pegue um bloco e o chame de hora em hora. Assim, se eu passar o bloco **`do puts 'DONG!' end`**, ele deve tocar como um relógio cuco. Teste seu método com diferentes blocos (inclusive o que eu mostrei para você). ***Dica:*** *Você pode usar* **`Time.now.hour`** *para pegar a hora atual. Porém, isso retorna um número entre* **`0`** *e* **`23`***, então você deve alterar esses números para os números de um relógio normal, entre (*`1` *e* `12`*).*

• *Logger do programa*. Escreva um método chamado **`log`**, que pegue uma string como descrição de um bloco e, é claro, um bloco. Similarmente ao **`FacaUmaCoisaImportante`**, essa deve retornar (**`puts`**) uma string dizendo que o bloco foi iniciado e outra string ao fim, dizendo que é o fim da execução do bloco, e também dizendo o que o bloco retornou. Teste seu método enviando um bloco de código. Dentro do bloco, coloque *outra* chamada para **`log`**, passando outro bloco para o mesmo (isto é chamado de *nesting*). Em outras palavras, sua saída deve se parecer com isso:

```
Começando "bloco externo"...
Começando "um bloco um pouco menor"...
..."um bloco um pouco menor" terminou retornando:  5
Começando "um outro bloco"...
..."um outro bloco" terminou retornando: Eu gosto de comida tailandesa!
..."bloco externo" terminou retornando: false
```

• *Um Logger aperfeiçoado*. A saída do último logger é muito difícil de ler, e fica muito pior a medida que você for usando. Seria muito mais fácil de ler se você identasse as linhas para os blocos internos. Para fazer isso, você vai precisar saber quão profundamente aninhado você está toda vez que for escrever algo no log. Para fazer isso, use uma *variável global*, uma variável que você possa ver de qualquer lugar de seu código. Para instânciar uma variável global, você deve precedê-la com um **`$`**, assim: **`$global`**, **`$nestingDepth`**, e **`$bigTopPeeWee`**. Enfim, seu logger deve ter uma saída parecida com essa:

```
Começando "bloco externo"...
  Começando "um pequeno bloco"...
    Começando "pequenino bloco"...
    ..."pequenino bloco" terminou retornando: muito amor
  ..."um pequeno bloco" terminou retornando: 42
  Começando "um outro bloco"...
  ..."um outro bloco" terminou retornando: Eu adoro comida indiana!
```

```
..."bloco externo" terminou retornando: true
```

Bem, isso é tudo que você aprendeu com esse tutorial. Parabéns! Você aprendeu *muito*. Talvez você sinta como se não lembrasse de nada, ou talvez você tenha pulado algumas partes... Relaxe. Programação não é o que você sabe, e sim o que você faz. À medida que você for aprendendo onde procurar as coisas que você esquecer, você estará indo bem. Eu espero que você não ache que eu escrevi tudo isso sem ficar conferindo a cada minuto! Porque eu fiz isso. Eu também tive muita ajuda com os códigos que rodam em todos os exemplos desse tutorial. Mas onde *eu* estava pesquisando tudo e a quem *eu* estava pedindo ajuda? Deixe-me conhecê-lo...

Aprenda a Programar

11. Além deste Tutorial

Então, onde é que podemos ir agora? Se você tiver uma pergunta, para quem pode perguntar? E se você quer que seu programa abra uma página da Web, envie um e-mail, ou redimensione uma foto digital? Pois bem, há muitos, muitos lugares onde encontrar ajuda para Ruby. Infelizmente, essa resposta não tem muita utilidade, não é? :-)

Para mim, realmente existem apenas três lugares onde procuro para ajuda para o Ruby. Se for uma pequena questão e eu acho que posso experimentar sozinho para encontrar a resposta, uso *irb*. Se for uma questão maior, procuro no meu *Pickaxe*. E se simplesmente não consigo dar conta do recado, então peço ajuda na lista *ruby-talk*.

IRB: Ruby Interativo

Se você instalou Ruby, então você instalou irb. Para usá-lo, basta ir ao seu prompt de comando e digitar `irb`. Quando você estiver em irb, você pode digitar qualquer expressão ruby que você queira, e ele devolverá o valor da mesma. Digite `1 + 2`, e devolverá `3`. (Note que você não precisa usar **`puts`**). É como uma espécie de calculadora Ruby gigante. Quando tiver concluído, digite simplesmente `exit`.

Há muito mais do que isso a respeito do irb, mas você pode aprender tudo sobre ele no Pickaxe.

O Pickaxe: "Programming Ruby"

O livro sobre Ruby que você não pode perder de jeito nenhum é "Programming Ruby, The Pragmatic Programmer's Guide", de Andrew Hunt e David Thomas (os Programadores Pragmáticos). Embora eu recomende fortemente a 2ª edição deste livro excelente, com a cobertura de todas as últimas novidades do Ruby, você também pode obter uma versão on-line grátis de pouco mais antiga (mas ainda relevante). (Na verdade, se você instalou a versão do Ruby para Windows, você já tem ela).

Você pode encontrar praticamente tudo sobre Ruby, do básico ao avançado, neste livro. É fácil de ler; é abrangente; é quase perfeito. Eu gostaria que todas as linguagens (de programação) tivessem um livro desse nível. Na parte final do livro, você encontrará uma enorme seção detalhando cada método de cada classe, explicando-o e dando exemplos. Eu simplesmente amo este livro!

Há um sem número de lugares onde você pode obtê-lo (incluindo o próprio site do Pragmatic Programmers), mas o meu lugar favorito é no ruby-doc.org. Esta versão tem um bom índice , bem como um índice remessivo (no ruby-doc.org tem muitas outras ótimas documentações, tais como a API Central (Core API) e a Biblioteca Padrão (Standard Library)... Basicamente, isso documenta tudo que vem com o Ruby. Verifique.).

E por que é chamado de "o Pickaxe" (picareta)? Pois bem, há uma imagem de uma picareta na capa do livro. É um nome bobo, eu acho, mas pegou.

Ruby-Talk: uma lista de discussão sobre Ruby

Mesmo com o irb e o Pickaxe, às vezes você ainda pode não dar conta sozinho. Ou talvez você queira saber se alguém já fez o que você está fazendo, para ver se você pode utilizá-lo. Nestes casos, o melhor lugar é ruby-talk, a lista de discussão do Ruby. É cheia de gente amigável, inteligente, colaborativa. Para saber mais sobre ela, ou para se inscrever, procure aqui.

ATENÇÃO: Existe um *grande* número de e-mails na lista todos os dias. Eu criei uma regra no meu cliente de e-mail para que não fique tudo na mesma pasta. Mas se você não quiser ter que fazer isso, você não precisa! A lista de discussão ruby-talk tem um espelho no grupo de notícias comp.lang.ruby, assim, você pode ver as mensagens por lá. Em suma, você verá as mesmas mensagens, mas de uma maneira um pouco diferente.

Tim Toady

Tenho tentado proteger você de algo com o que vai esbarrar em breve, é o conceito de TMTOWTDI (pronunciado como "Tim Toady"): There's More Than One Way To Do It (Há Mais De Uma Maneira Para Fazer Isso).

Agora alguns vão lhe dizer que TMTOWTDI é uma coisa maravilhosa, enquanto outros se sentem bastante diferente. Eu realmente não tenho fortes sentimentos sobre o assunto em geral, mas eu acho que é uma péssima maneira de ensinar para alguém como programar. (Como se aprender uma maneira de fazer alguma coisa não fosse suficientemente desafiante e confuso!).

No entanto, agora que você está indo além deste tutorial, você verá muito código diversificado. Por exemplo, posso pensar em pelo menos cinco outras maneiras de criar uma string (além de circundar algum texto com aspas simples), e cada um deles funciona de maneira um pouco diferente. Eu só lhe mostrei o mais simples dos seis.

E quando conversamos sobre ramificações, mostrei para você **`if`**, mas não lhe mostrei **`unless`**. Eu vou deixá-lo entender isso no irb.

Outro agradável atalho que você pode usar com **`if`**, **`unless`**, **`while`**, é a elegante versão de uma linha:

```
#  Estas palavras são de um programa que escrevi para gerar
#  baboseiras em inglês. Bacana, não?
puts 'grobably combergearl kitatently thememberate' if 5 == 2**2 + 1**1
puts 'enlestrationshifter supposine follutify blace' unless 'Chris'.length == 5
```

```
grobably combergearl kitatently thememberate
```

E, finalmente, há uma outra forma de escrever métodos que recebem blocos (não procs). Vimos isso onde recebemos o bloco e o transformamos em uma proc usando o truque do **`&block`** na lista de parâmetros quando se define a função. Depois, para chamar o bloco, você usa **`block.call`**. Pois bem, há um caminho mais curto (embora pessoalmente eu o julgue mais confuso). Em vez disto:

```
def facaDuasVezes(&bloco)
  bloco.call
  bloco.call
end

facaDuasVezes do
  puts 'murditivent flavitemphan siresent litics'
end
```

```
murditivent flavitemphan siresent litics
murditivent flavitemphan siresent litics
```

...você faz isso:

```
def facaDuasVezes
  yield
  yield
end

facaDuasVezes do
  puts 'buritiate mustripe lablic acticise'
end
```

```
buritiate mustripe lablic acticise
buritiate mustripe lablic acticise
```

Não sei ... O que você acha? Talvez seja eu, mas... **`yield`**?! Se fosse algo como **`chame_o_bloco_escondido`** (**`call_the_hidden_block`**) ou algo assim, faria *muito* mais sentido para mim. Muitas pessoas dizem que **`yield`** faz sentido para elas. Mas acho que esse é o propósito de TMTOWTDI: elas fazem do jeito delas, e eu vou fazer à minha maneira.

O FIM

Use-o para o bem e não para o mal. :-) E se você achou este tutorial útil (ou confuso, ou se você encontrou um erro), me faça saber!