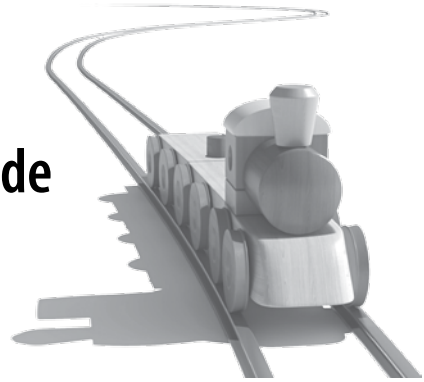
CAPÍTULO 1

Instalando o ambiente de desenvolvimento



Antes de começarmos nosso estudo de Rails, é importante instalar o ambiente necessário. Para obter melhor proveito deste livro, você precisará dos seguintes itens:

- Uma instalação do Ruby 1.9 (a versão 1.8.7 deve funcionar, mas será necessário converter para a nova sintaxe de hashes, que é utilizada em vários pontos do livro).
- Rubygems, em uma versão recente – no mínimo, 1.8.
- Rails instalado com suporte a SQLite3.
- Um bom editor de textos.

Os passos para instalação de cada um desses itens varia um pouco, dependendo de qual plataforma você usa: Mac, Linux ou Windows. Vamos então descrever um pouquinho de cada um desses itens antes de entrar na introdução ao Ruby.

Editor de textos

Você vai precisar de um bom editor de textos para desenvolver aplicações em Rails. Existem também muitas IDEs, e você pode utilizá-las. Vou listar algumas opções. É importante que você utilize a que mais lhe agrade. Não existe uma escolha certa ou errada.

Na edição anterior do livro, havia uma descrição detalhada de diversas IDEs e editores de texto muito bons, mas essa descrição detalhada fica desatualizada tão rápido que quando o livro chegou às prateleiras as descrições já não estavam tão corretas. Então vamos mudar a abordagem, e desta vez vou inserir breves comentários com minhas opiniões e links para o site do editor ou IDE, mas lembrem-se: é a minha opinião sobre o editor, isso não quer dizer que você deva concordar.

Multiplataforma

VIM

É o meu editor preferido. É o ambiente de desenvolvimento que eu uso enquanto estou trabalhando com Ruby. Existem diversos plugins disponíveis na web para adicionar os mais diversos recursos ao VIM. Possui versões para todas as plataformas conhecidas e tem um desempenho excelente. Se você for utilizar o VIM, aproveite o projeto "vimfiles" no meu github, que irá instalar diversos plugins e deixar o seu VIM prontinho para o trabalho.

Site para download: `http://www.vim.org`

Rubymine

É uma excelente IDE para o desenvolvimento de aplicações em Ruby e Rails, e foi criada sobre a base do IntelliJ IDEA, a melhor IDE paga para desenvolvimento em Java. Também tem diversos plugins e proporciona um ambiente totalmente integrado para o desenvolvimento, controle de versões, testes e depuração de aplicações. Considero o suporte a debug passo a passo simplesmente espetacular – ele me salvou algumas vezes.

Site para download: `http://www.jetbrains.com/ruby/`

Aptana RadRails

É uma IDE para desenvolvimento em Ruby e Rails escrita sobre o Eclipse, que é a melhor IDE gratuita para desenvolvimento Java que eu conheço. Tem suporte a todos os recursos do Rubymine, com algumas diferenças de implementação. Uma boa pedida para quem está acostumado com o Eclipse.

Site para download: `http://www.aptana.com/products/radrails`

Sublime Text2

Disponível para Linux, Windows e Mac. É um excelente editor de textos, com alguns recursos bastante interessantes e suporte a bundles do TextMate. Testei por algum tempo e o achei bastante produtivo e leve. Vale o teste para quem usa Mac e alguma outra plataforma e se sente órfão de TextMate quando não está no Mac, ou se você é obrigado a usar Windows. É pouco provável achar um editor muito melhor, exceto o VIM.

Site para download: `http://www.sublimetext.com/2`

Linux

GEdit

GEdit é o editor padrão do Gnome, e com alguns plug-ins se torna um bom editor para trabalhar com Ruby. Gosto bastante da possibilidade de mostrar um terminal na parte de baixo da tela e da extensibilidade e facilidade de criar plugins. O problema é que os plugins são escritos em Python, e não em Ruby.

Site para download: `http://projects.gnome.org/gedit/`

Windows

Notepad++

É um editor de textos bastante simples, mas tem sintax highlight para diversas linguagens. A vantagem é que ele é extremamente leve e há uma grande chance de o seu colega desenvolvedor já ter uma cópia dele instalado.

Site para download: `http://notepad-plus-plus.org/`

E-Texteditor

Este editor foi criado com o objetivo de ser o "TextMate" para Windows – e conseguiu. Ele é um pouco mais pesado, mas tem a interface parecida com o TextMate. Suporta todos os bundles e trabalha integrado com o Cygwin, para você não perder o poder do Unix, mesmo estando preso ao Windows. Existe uma lenda de que é possível compilar ele para Linux, mas eu não tive a capacidade ou a determinação necessárias para testar isso.

Site para download: `http://www.e-texteditor.com/`

Mac

TextMate

Este é o editor padrão para desenvolvedores que trabalham com Mac. Entre os que conheço, ou trabalham com TextMate ou com VIM. Existem muitos bundles disponibilizando recursos novos para este editor, e muitas pessoas bastante famosas na comunidade Ruby gostam bastante dele. Eu não consegui ver muitas vantagens nele, mas recomendo que faça um teste e forme a sua própria opinião.

Site para download: `http://macromates.com/`

Instalando o Ruby

Agora que você já escolheu um editor de textos, pode começar a programar em Ruby, e para isso precisamos ter o Ruby instalado.

Vou descrever como instalar o Ruby e o Rails para cada plataforma. O resultado não será exatamente o mesmo, mas você poderá aproveitar bastante este livro independentemente da plataforma que utilize. Quando começar a trabalhar com Rails, para que possa ser realmente produtivo, sugiro uma plataforma baseada em Unix, ou seja, use Linux ou Mac, pois o Windows não vai colaborar muito. Diversas coisas que funcionam bem em outras plataformas não funcionam bem no Windows.

Todos os exemplos do livro foram testados no Mac e no Linux. Se não for possível instalar algum pacote no Windows, por favor, entre em contato, que tentarei procurar alguma forma de ajudar.

Windows

Para instalar todo o ambiente no Windows, a tarefa será bastante fácil com o RailsInstaller, criado pelo pessoal da Engine Yard. Baixe o pacote em `http://railsinstaller.org/` e siga o assistente de instalação; depois de alguns cliques em "Next", você terá um ambiente com os seguintes softwares:

- Ruby 1.9.2
- Bundler 1.0.18
- Rails 3.1.1

- Git 1.7.6
- Sqlite 3.7.3

Ou seja, tudo o que precisaremos para os exercícios do livro.

Linux

A maior parte das distribuições Linux tem pacotes de Ruby para serem instalados. Os comandos a seguir funcionarão para uma versão recente do Ubuntu. Se você usa outra distribuição, dê uma olhada na ferramenta de gerenciamento de pacotes da sua distribuição.

```
sudo apt-get install ruby1.9 irb1.9 ruby1.9-dev git-core libsqlite3-dev sqlite3
wget http://production.cf.rubygems.org/rubygems/rubygems-1.8.11.tgz
tar -xzf rubygems-1.8.11.tgz
cd rubygems-1.8.11
sudo ruby1.9 setup.rb
sudo ln -sf /usr/bin/ruby1.9 /usr/bin/ruby
sudo ln -sf /usr/bin/gem1.9 /usr/bin/gem
gem install rails
```

Mac

O Mac vem com uma versão do Ruby instalada, mas é uma versão antiga. O processo de instalação mais fácil é realizado utilizando-se o RVM. Então o processo que vou descrever também funcionará em um Linux.

Para instalar o RVM, o Ruby 1.9.2 e o Rails, utilize os seguintes comandos:

```
bash < <(curl -s https://raw.github.com/wayneeseguin/rvm/master/binscripts/rvm-installer )
echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm" # Load RVM function'
>> ~/.profile
source "$HOME/.rvm/scripts/rvm"
rvm install 1.9.2
rvm use 1.9.2 --default
gem install rails
```

Para que esses comandos funcionem, você precisa ter o Xcode instalado (o Xcode está disponível na Mac Apple Store). Se você não vai precisar do Xcode completo, pode tentar instalar apenas o GCC para Mac, a partir deste endereço: `https://github.com/kennethreitz/osx-gcc-installer`.

O que é, por que e como é o Ruby

Não, você não está lendo o livro errado, este livro é sobre Ruby on Rails, mas este primeiro capítulo apresentará uma introdução ao Ruby, já que o Rails é escrito em Ruby, e é necessário aprender pelo menos um pouco de Ruby para utilizar bem o Rails.

Depois que você se familiarizar mais com o Rails, sugiro que estude mais a fundo a linguagem Ruby, o que vai permitir que você descubra muitas outras ferramentas bastante úteis, como EventMachine, que é uma ótima linguagem para automação de processos e devops em servidores Unix, e até mesmo para escrever parte de sua aplicação .NET. Mas vamos focar no que interessa: aprender um pouco de Ruby para que possamos utilizar bem o Rails.

Interactive Ruby

No início desta jornada pelo Ruby, vamos utilizar o `irb`, sigla em inglês para "Interactive Ruby" (Ruby Interativo). Esse utilitário é muito usado para testar códigos antes de passá-los para o arquivo de destino, permitindo que o teste seja mais rápido e, portanto, que a correção dos possíveis problemas também seja mais rápida.

Abra um terminal e digite `irb`. Você verá um prompt como este:

```
irb(main):001:0>
```

Agora, você está pronto para começar a utilizar o Ruby como uma calculadora!

Lembre-se de que todos os comandos mostrados nas próximas páginas podem ser testados neste aplicativo. Então, vamos aprender um pouco sobre o irb antes de continuar.

Se você digitar `1+1` e pressionar Enter, na próxima linha vai aparecer:

```
=> 2
```

e o prompt mudará para:

```
irb(main):002:0>
```

Ou seja, os números entre os dois pontos são correspondentes à linha do comando, como se fosse a linha do arquivo na edição de um arquivo `.rb` (extensão padrão para programas Ruby). Também podemos concluir que toda

e qualquer expressão em Ruby tem um valor de retorno, que é mostrado no irb, no caso o `=> 2`, que é o resultado da soma realizada.

O último número é o nível do bloco atual.

No Ruby, tudo é um objeto, incluindo números. O que fizemos neste exemplo foi enviar a mensagem `+` para o objeto `1`, com o argumento `1`, e essa mensagem retorna um resultado que é a soma desses números – no caso, 2. Esse conceito de enviar mensagens para objetos será muito importante mais adiante.

Introdução ao Ruby

Depois de todos os códigos terem sido apresentados, você poderá escrever linha a linha no IRB para ver o que acontece a cada linha. Para facilitar a escrita do tutorial e a posterior atualização do mesmo, vou colocar todos os códigos em arquivos .rb; e, para executar esses arquivos, salve o arquivo e execute `ruby nome_arquivo.rb` . Você verá o resultado da execução no console.

Vamos começar com o básico do básico, como definir um método no Ruby. Na verdade, não estamos definindo um método, estamos registrando um endereço para que o carteiro interno do Ruby possa entregar mensagens a um objeto. Estamos definindo os nomes das mensagens que esse objeto pode receber, mas, para fazer isso, utiliza-se a palavra `def`, como no exemplo a seguir:

▸ intro/01soma.rb

```
def soma a, b
    a + b
end

puts soma 1, 2
```

No Ruby não é necessário utilizar `return` ou qualquer palavra-chave para definir o retorno de um método. O retorno do método é o valor da última expressão executada; em nosso caso, `a+b`.

Os parênteses são opcionais em quase todas as situações. Por exemplo, poderíamos reescrever o exemplo anterior da seguinte forma:

▸ intro/02soma.rb

```
def soma(a, b)
    a + b
end

puts soma(1, 2)
```

Como podemos verificar no exemplo, além dos parênteses, o caractere ; no final das sentenças também é opcional, tornando a sintaxe bastante flexível.

No Ruby, tudo é um objeto, e todo objeto pertence a uma classe. Todas as classes descendem de `Object`, e é possível verificar em tempo de execução qual o tipo do objeto, como pode ser visto a seguir:

▸ intro/03objects.rb

```
puts 1.class
puts self.class
puts [].class
puts ({}.class)
puts "a".class
puts 1.1.class
puts 99999999999999999999.class
```

A palavra-chave `self` é utilizada para identificar o objeto atual, o número `1` é do tipo `Fixnum`, o número `1.1` é do tipo `Float`, e um número muito grande é do tipo `Bignum`. Não é necessário se preocupar se um número é `Fixnum` ou `Bignum`, pois o Ruby vai cuidar das conversões automaticamente.

Ok, mas em que saber isso pode ser útil? Fácil: a classe `Fixnum`, por consequência, a instância dessa classe `1`, tem diversos métodos úteis para trabalhar com números, e, graças ao irb, não precisamos decorar todos os métodos.

▸ intro/04methods.rb

```
puts 1.methods.sort
```

A classe `Object` tem o método `methods`, que retorna um Array contendo todos os métodos disponíveis para aquele objeto; neste caso, o número 1.

Como podemos verificar pelo resultado do método `methods`, as operações matemáticas também são métodos da classe `Fixnum`, como o `+`, `*`, `-`, `/` e assim por diante.

Classes abertas em tempo de execução

O Ruby tem um recurso bastante interessante, que o torna uma linguagem muito flexível. Esse recurso se chama "open classes", ou seja, todas as classes do Ruby estão sempre abertas, o que possibilita que elas sejam alteradas a qualquer momento. Isso é bastante útil, mas também bastante perigoso. Para demonstrar o perigo que isso representa em mãos incautas, faremos uma pequena brincadeira:

▸ intro/05openclass.rb

```
class Fixnum
    def +(other)
        self - other
    end
end
puts 1 + 5
```

O que acabamos de fazer foi alterar o método `+` da classe `Fixnum`, o que pode causar uma grande confusão, como visto no exemplo, fazendo com que o método `+` realize, na verdade, uma operação de subtração. No entanto, esse recurso, quando utilizado corretamente, pode ajudar bastante no desenvolvimento de aplicações, como veremos mais adiante.

Variáveis e escopo

No Ruby, não é necessário declarar uma variável: ela será definida no momento em que tiver um valor atribuído. Para que isso seja possível, o Ruby utiliza tipagem dinâmica, ou seja, ele define o tipo de uma variável por seu valor, mas isso não quer dizer que seja uma linguagem de tipagem fraca, pois não é possível somar um número com uma string, como pode ser visto no seguinte código-fonte:

▸ intro/06strongtype.rb

```
puts 1 + "2"
```

Se você executar esse código, receberá um erro informando que não é possível misturar variáveis de tipos diferentes.

O Ruby não tem palavras-chave para definir o escopo de variáveis: isso é feito por meio de símbolos, como na tabela a seguir:

Símbolo	Descrição
`nome`	Variável local.
`@nome`	Variável de instância.
`@@nome`	Variável de classe.
`$nome`	Variável global.

Tipos básicos do Ruby

Blocos de código

Blocos de código consistem em um dos recursos mais versáteis e flexíveis do Ruby. São utilizados para iterar em coleções, personalizar o comportamento de algum método e definir "Domain Specific Languages". Para quase tudo são usados blocos de código, até para criar formulários para páginas da web no Rails e definir boa parte das configurações do framework.

Existem duas sintaxes diferentes para definir um bloco de código. Pode-se fazer isso utilizando os símbolos `{` e `}` ou utilizando as palavras-chave `do` e `end`.

Blocos de código podem receber parâmetros. Para definir a lista de parâmetros, logo depois de aberto o bloco de código é utilizado o símbolo `|` para demarcar o início e o fim da lista de parâmetros.

O Ruby tem suporte a closures reais. Isso quer dizer que, se em um bloco de código forem utilizadas variáveis visíveis no contexto da criação do bloco, qualquer alteração nessas variáveis será refletida no contexto original, ou seja, os blocos de código têm um link com o contexto de origem, o que os torna mais úteis ainda e os diferencia muito de ponteiros para métodos do C/C++, por exemplo.

▶ intro/07codeblock.rb

```
arr = [1, 2, 3, 4]
arr.each { |val|
    print "#{val}\n"
}
```

Esse exemplo utiliza um bloco de código simples, delimitado por chaves para iterar nos valores de um array. Veja que o valor do array não foi alterado pelo retorno da expressão.

▸ intro/08codeblock.rb

```
arr = [1, 2, 3, 4]
arr.each_with_index do |val, idx|
    print "Posicao #{idx} valor #{val}\n"
end
```

Esse segundo exemplo faz praticamente a mesma coisa, mas utiliza outro método de iteração, que passa além do valor. O índice fornece o valor no array e, dessa vez, são utilizados os delimitadores `do` e `end`.

▸ intro/09closure.rb

```
arr = [1, 2, 3, 4]
valor = 1
arr.each do |val|
    valor += val
end

puts valor
```

Já nesse exemplo é utilizada uma closure, em vez de um bloco de código simples. A diferença básica é que a closure mantém o contexto de onde ela é chamada, neste caso, alterando a variável local valor. Podemos ver no próximo exemplo que, se for utilizado um método para realizar essa iteração, a variável valor não vai existir, o que vai causar um erro que prova que o bloco aponta para seu contexto de origem.

▸ intro/15scope.rb

```
valor = 1
def iterar
    arr = [1, 2, 3, 4]
    arr.each do |val|
        valor += val
    end
end

iterar
```

Esse exemplo também demonstra uma das convenções para a nomenclatura de variáveis apresentada no exemplo anterior, ou seja, variáveis que não têm seus nomes iniciados por `@`, `@@` ou `$` são variáveis locais no contexto onde foram definidas.

É possível também passar blocos de código como parâmetros para métodos. Isso é muito usado na API padrão do Ruby, e pode ser visto inclusive no exemplo anterior, quando passamos um bloco de código como parâmetro para o método `each` da classe `Array`. No próximo exemplo, veremos como criar um método que recebe um bloco de código como parâmetro, ou seja, como utilizar um bloco de código passado para um método. Não será mostrado como armazenar um bloco de código em uma variável, porque, nesse caso, o bloco de código se transforma em uma instância da classe `Proc`, e esse é um assunto para o próximo capítulo.

▸ intro/16blockparam.rb

```
def recebe_proc_e_passa_parametro
    if block_given?
        yield
    else
        puts "você precisa passar um bloco para este método\n"
    end
end

recebe_proc_e_passa_parametro
recebe_proc_e_passa_parametro { print "dentro do bloco\n" }
```

Podemos também passar parâmetros para blocos recebidos nos métodos:

▸ intro/17blockparam.rb

```
#encoding : UTF-8
def recebe_proc_e_passa_parametro
    if block_given?
        yield(23)
    else
        puts "você precisa passar um bloco para este método\n"
    end
end
recebe_proc_e_passa_parametro do |par|
    puts "Recebi #{par} dentro deste bloco\n"
end
```

Procs

Procs são muito parecidos com blocos de código ou closures simples, que foram vistos na seção anterior. A diferença básica é que podemos armazenar um proc em uma variável, e isso torna um pouco mais "caro" para o Ruby a implementação de procs do que simples blocos de código. Por isso, tiveram a brilhante ideia de, internamente no Ruby, utilizar dois elementos diferentes para representar essas duas entidades que, para o programador usuário da linguagem, são praticamente iguais.

Além disso, o Ruby converte blocos de código em procs de forma transparente, fazendo com que na maioria das situações não seja necessário se preocupar com esse tipo de detalhe. Para deixar isso um pouco mais claro, veremos alguns exemplos a seguir:

Converter um bloco recebido em um proc

▸ intro/18proc.rb

```
def recebe_proc(&block)
    if block
        block.call
    end
end

recebe_proc
recebe_proc { print "este bloco vai se tornar uma proc pois vai ser atribuído a uma
variável no método" }
```

Criar uma variável do tipo `Proc` de duas formas diferentes

▸ intro/19proc.rb

```
p = Proc.new { print "este bloco vai se tornar uma proc pois está sendo atribuído a uma
variável\n" }
p.call
```

▸ intro/20lambda.rb

```
p1 = lambda do
    print "este bloco vai se tornar uma proc pois está sendo atribuído a uma variável\n"
end

p1.call
```

Nesses exemplos, foi possível verificar as duas formas de criar um objeto do tipo `Proc`: utilizando o construtor da classe `Proc` (leia mais sobre construtores na seção sobre classes) ou utilizando a palavra-chave `lambda`, de modo semelhante ao Python.

Como `Proc` é uma classe, objetos desse tipo têm diversos métodos. Vejamos alguns desses de forma mais detalhada:

▸ intro/21procclass.rb

```
p = Proc.new { print "a" }
puts p.methods.sort
```

Diversos métodos padrão da classe `Object` do Ruby são listados, pois, como todas as classes, a classe `Proc` também é descendente de `Object`, ou seja, procs são objetos comuns no Ruby. Retomando o tópico anterior, existem alguns métodos que são mais interessantes para nós neste momento:

Método	Descrição
`call`	Utilizado para executar o proc, os parâmetros que forem definidos no bloco são passados como parâmetros para o método `call`.
`[]`	Alias para o método `call`, ou seja, podemos executar um proc com a sintaxe: `p[`*`parâmetros`*`]`.
`arity`	Informa o número de parâmetros definidos nesse proc.
`binding`	Retorna a binding correspondente ao local onde foi definido o bloco de código que deu origem a esse proc.

Números

Como em todas as linguagens de programação, o Ruby tem suporte a processamento numérico, mas, diferente de muitas das linguagens, mesmo algumas das supostamente orientadas a objetos, no Ruby, todos os números são objetos, portanto, todos são instâncias de alguma classe.

O Ruby tem três classes que representam números. São elas:

Classe	Descrição
`Fixnum`	Representa inteiros de -1073741824 a 1073741823.
`Bignum`	Representa inteiros fora do intervalo da classe `Fixnum`.
`Float`	Representa números de ponto flutuante.

Para definir um número de ponto flutuante, basta utilizar um ponto `.`. Se você quiser muito separar milhares, pode utilizar o `_`, mas não recomendo a utilização, pois o visual da sintaxe fica poluído.

A seguir estão alguns exemplos de números no Ruby.

▸ intro/22numberandclass.rb

```
def number_and_class(n)
    puts "#{n} -> #{n.class}"
end

i = 1
i1 = 1.1
i2 = 111_222_333
i3 = 999999999999999999

number_and_class i
number_and_class i1
number_and_class i2
number_and_class i3
```

Como já foi dito, o Ruby utiliza duas classes diferentes para tratar inteiros. Não é necessário se preocupar com isso, pois a conversão entre esses tipos é feita automaticamente pelo interpretador, ou seja, se um número ficar grande demais para caber em um `Fixnum`, ele será convertido para `Bignum`. Caso ele volte a estar no intervalo do `Fixnum`, essa classe será utilizada novamente.

Valores booleanos

Valores booleanos, como na maior parte das linguagens de programação, são representados por `true` e `false`, ou verdadeiro e falso, respectivamente. Entretanto, no Ruby existe uma particularidade: a palavra-chave `nil`, que representa "nenhum objeto", é considerada como falsa em qualquer comparação. Portanto, existem dois valores para falso, e quaisquer outros valores serão considerados como verdadeiros. Resumindo: as comparações booleanas servem também para verificar se uma variável tem um valor.

Veja alguns exemplos que demonstram melhor o que você acabou de ler:

▸ intro/23boolean.rb

```
#encoding: UTF-8
def testa_valor(val)
    if val
        print "#{val} é considerado verdadeiro pelo Ruby\n"
    else
```

```
        print "#{val} é considerado falso pelo Ruby\n"
    end
end

testa_valor true

testa_valor false

testa_valor 1

testa_valor "asda"

testa_valor nil
```

Strings

Strings são o tipo de dado utilizado para representar texto dentro do código. O Ruby tem dois tipos de string, que são diferentes apenas durante sua declaração. Um dos tipos tem expansão de variáveis e suporta caracteres especiais como os do C; por exemplo, `\n` para representar uma quebra de linha.

Além desses dois tipos de strings, o Ruby tem diversas formas de declarar strings no código, como pode ser visto na tabela a seguir:

Símbolo	Descrição
aspas	String simples com expansão de variáveis.
apóstrofes	String simples sem expansão de variáveis.
<<MARCADOR	String multilinha com expansão de variáveis.
%Q{ }	String multilinha com expansão de variáveis.
%q{ }	String multilinha sem expansão de variáveis.

Veja alguns exemplos de declaração de strings no Ruby:

▶ intro/24strings.rb

```
a = "texto"
b = 'texto'
c = "texto\nsegunda linha"
d = 'texto\nmesma linha'
e = "a = #{a} - é assim que se utiliza expansão de variáveis"
f = <<__ATE_O_FINAL
esta é
uma String
bem grande e só tesmina
quando encontrar o marcador __ATE_O_FINAL
```

```
no início de uma linha
__ATE_O_FINAL
g = %Q{Esta também
é uma String
com mais de uma linha
e também suporta #{a}
expansão de variáveis
}
h = %q{Já
esta
que também é multi linha
não suporta #{a}
expansão de variáveis}
puts a
puts b
puts c
puts d
puts e
puts f
puts g
puts h
```

Como podemos ver no exemplo anterior, as strings do Ruby suportam um recurso bastante interessante chamado expansão de variáveis, que, na verdade, suporta execução de código Ruby dentro de uma String. Para utilizar isso, basta colocar o código a ser executado dentro dos símbolos `#{` e `}`. O código será executado, e o resultado da expressão será incluído dentro do texto original.

Isso é bastante útil ao se trabalhar com output de texto, seja para exibição aos usuários, seja para criação de arquivos.

Constantes

O Ruby não tem algo que seja realmente constante, mas a linguagem tem um padrão que diz que variáveis declaradas com a primeira letra maiúscula são constantes. A linguagem não o impede de alterar o valor dessa constante se você realmente quiser: o Ruby vai apenas imprimir um aviso dizendo que você realmente não deveria estar fazendo isso, como mostra o exemplo a seguir:

▶ intro/25constants.rb

```
CONSTANTE = "asda"
CONSTANTE = 1
Constante = 2
Constante = 5
```

Intervalos numéricos

Além de números simples, é possível declarar intervalos numéricos no Ruby. O Ruby tem dois tipos de intervalos numéricos: o inclusivo, que inclui o último número, e o exclusivo, que não inclui esse número. Intervalos inclusivos e exclusivos são declarados utilizando os símbolos ".." e "...", respectivamente, como pode ser visto no exemplo a seguir:

▶ intro/26numericintervals.rb

```
a = 1..10
b = 1...10
a.each do |v|
    print "#{v} "
end
print "\n"
b.each do |v|
    print "#{v} "
end
```

Arrays

Arrays são coleções de valores. No Ruby, arrays não são tipados, ou seja, um array pode conter objetos de diversos tipos. Existem duas formas de se declarar um array genérico e uma forma extra de se declarar um array contendo apenas strings, como podemos ver no exemplo a seguir:

▶ intro/27arrays.rb

```
arr = []
arr = ['a', 1]
arr = Array.new
arr = %w{ a b c }
puts arr.methods.sort
```

Junto com os hashes, arrays são as estruturas de dados mais utilizadas na linguagem Ruby, principalmente pela flexibilidade e facilidade de iteração pelo conteúdo de um array. Como vimos nos outros exemplos, é bastante fácil percorrer um array utilizando o método `each`, ou `each_with_index`. Além desses, os arrays têm diversos outros métodos que serão muito úteis durante o desenvolvimento das aplicações Rails. Alguns desses estão na tabela a seguir:

Método	Descrição
`select`	Recebe um bloco e retorna um novo array contendo todos os elementos para os quais o bloco retornou true.
`[]=`	Define o valor de uma posição no array.
`[]`	Retorna o valor da posição passada como parâmetro.
`last`	Retorna o último item do array.
`empty?`	Retorna verdadeiro se o array estiver vazio.
`equal?`	Compara com outro array.
`each_index`	Recebe um bloco e passa apenas os índices do array para o bloco.
`sort`	Retorna um novo array contendo os itens deste ordenados. Opcionalmente, recebe um bloco com dois parâmetros, o qual deve retornar qual dos itens é menor, fazendo assim uma comparação personalizada.
`sort!`	Similar ao `sort`, mas altera o array de origem.
`+`	Soma dois arrays, criando um novo com os itens de ambos.
`-`	Subtrai dois arrays, criando um novo com os itens do primeiro não contidos no segundo.
`push`	Adiciona um item no final do array.
`pop`	Retorna o último item do array e o remove do array.
`find`	Recebe um bloco com um parâmetro e retorna o primeiro item para o qual o bloco retornar verdadeiro.
`clear`	Remove todos os itens do array.
`shift`	Retorna o primeiro item do array e o remove do array.
`first`	Retorna o primeiro item do array.
`inject`	Recebe um valor inicial e um bloco com dois parâmetros. O primeiro é o valor atual, e o segundo, o item atual do array, retorna o resultado da operação realizada no bloco. O próximo exemplo utiliza o método inject para somar todos os itens de um array numérico.

▸ intro/28inject.rb

```
arr = [1, 2, 3, 4, 5, 6]
v = arr.inject(0) do |val, it|
    val + it
end

puts v
```

Existe ainda mais uma forma de criar um array no Ruby, que serve para criar métodos com uma lista variável de parâmetros. Para isso, basta declarar o último parâmetro de um método com o nome `*nome`, de forma que esse parâmetro seja um array contendo todos os parâmetros não declarados passados ao método, como pode ser visto no exemplo a seguir.

▶ intro/29varargs.rb

```
def parametros_variaveis(*arr)
    if arr
        arr.each do |v|
            print "#{v.class} - #{v}\n"
        end
    end
end

parametros_variaveis

parametros_variaveis 1, "asda", :simb, :a => "teste", :arr => %w{a b c}
```

Hashes

Hashes representam outra construção bastante utilizada no código Ruby. Utilizaremos muitos hashes no código Rails que será escrito nos próximos capítulos.

Hashes são semelhantes a arrays, mas não são simples coleções de objetos: são coleções do tipo `chave=valor`. Eles são tão comuns no código Ruby que existe até um operador especial para declarar hashes, que é o `=>`, que significa associado, ou seja, o valor à esquerda está associado ao valor à direita.

Existem duas formas de se declarar um hash. A primeira é utilizando o atalho `{}`, e a segunda, utilizando o construtor da classe `Hash`, como no exemplo a seguir:

▶ intro/30hashes.rb

```
h = {1 => "asda", "b" => 123}
h1 = {}
h2 = Hash.new
puts h.methods.sort
```

Hashes também têm alguns métodos que podem facilitar muito a vida dos programadores Ruby, como pode ser visto na tabela a seguir.

Método	Descrição
`[]`	Retorna o valor da chave passada como parâmetro.
`[]=`	Atribui o valor da chave.
`each`	Executa um bloco com dois argumentos para cada posição do mapa.
`each_key`	Executa um bloco com um argumento (a chave) para cada posição do mapa.
`each_value`	Executa um bloco com um argumento (o valor) para cada posição do mapa.
`has_key?`	Retorna verdadeiro se a chave existe no mapa.
`has_value?`	Retorna verdadeiro se o valor corresponde a alguma das chaves do mapa.
`Default=`	Possibilita configurar qual valor o mapa vai retornar quando for buscado o valor para uma chave inexistente.
`default_proc`	Idem a `default=`, mas executa um bloco para criar o valor para as novas chaves.
`delete`	Remove o item correspondente à chave indicada do mapa, retornando o valor da chave.

Símbolos

Símbolos, no Ruby, são aqueles nomes malucos que começam com "`:`" por todo o código. São muito utilizados como chaves em hashes e em quaisquer lugares onde você precisar de um rótulo (label) para alguma coisa. Eles são basicamente strings, mas ocupam muito menos processamento do interpretador Ruby e muito menos memória do que strings normais.

Strings podem ser transformadas em símbolos utilizando o método `to_sym`.

Toda vez em que for definir chaves em hashes que servirão como nomes de parâmetros a métodos, ou precisar enviar uma mensagem a um objeto, um símbolo será sempre melhor do que uma string, pois utilizará muito menos memória, o que fará com que sua aplicação gaste menos recursos e se comporte de maneira mais sociável com o computador.

Expressões regulares

Expressões regulares consistem na forma mais fácil de extrair informações de um texto ou alterar textos com padrões razoavelmente complexos.

O Ruby tem duas formas de declarar uma expressão regular e diversos métodos que recebem expressões regulares como parâmetros.

Uma coisa a ser lembrada, e que facilita muito a vida dos programadores Ruby, é que expressões regulares fazem parte da linguagem no Ruby. Isso facilita muito seu uso em comparação às linguagens em que expressões regulares são implementadas em forma de uma biblioteca externa.

As três formas de declarar uma expressão regular no Ruby são `/ER/`, `%r{ER}` ou por meio do método `new` da classe `Regexp`, como no exemplo a seguir:

▶ intro/31regexp.rb

```
er = /(.*?) .*/
er = %r{(.*?) .*}
er = Regexp.new "(.*?) .*"
er = /^[0-9]/
puts "123" =~ er
puts er =~ "123"
puts er =~ "abc"
puts er !~ "123"
puts er !~ "abc"
mt = /(..)\/(..)\/(....)/.match("12/05/2000")
puts mt.length
puts mt[0]
puts mt[1]
puts mt[2]
puts mt[3]
todo, dia, mes, ano = *(/(..)\/(..)\/(....)/.match("12/05/2000"))
puts todo
puts dia
puts mes
puts ano
puts "Urubatan".gsub(/ru/, "RU")
re = /.*/
puts re.methods.sort
```

Toda expressão regular é uma instância da classe `Regexp` e, da mesma forma que os números, a classe `Regexp` disponibiliza diversos métodos e operadores para facilitar as operações com expressões regulares, como podemos ver no exemplo a seguir.

Alguns métodos importantes disponíveis na classe `Regexp` são:

Método	Descrição
`=~`	Procura pela expressão regular no texto e retorna o índice em que ela foi encontrada.
`!~`	Informa se existe uma ocorrência da expressão regular no texto.
`match`	Retorna um objeto do tipo MatchData, que contém ponteiros para os locais onde cada grupo da expressão regular foi encontrado.

O operador * do Ruby utilizado no exemplo, quando usado em um array, expande o array em variáveis. Dessa forma pode-se atribuir um array a uma quantidade de variáveis igual às posições do array.

O método `gsub` da classe `String` demonstrado faz substituição de texto utilizando expressões regulares. É possível utilizar grupos na substituição, tornando o método ainda mais flexível.

Classes e métodos

Classes representam uma das bases da orientação a objetos no Ruby. Tudo no Ruby é um objeto, e todo objeto no Ruby é instância de uma classe. Por exemplo, `1` é uma instância da classe `Fixnum`, e todos os métodos dessa classe podem ser chamados nessa instância.

Assim, classes também são objetos no Ruby, portanto, podem ter métodos próprios, ou seja, é possível definir métodos de classe, diferentemente do Java ou do C++, que têm métodos estáticos. Métodos de classe são herdados por classes descendentes da classe onde foram definidos e podem saber a qual objeto pertencem, pois nessa situação a palavra-chave `self` vai apontar para a classe, e não para uma de suas instâncias.

Para definir uma classe é usada a palavra-chave `class`, seguida por um nome de uma constante, que será utilizado para referenciar aquela classe.

Variáveis de instância são definidas por meio da nomenclatura `@nome` e variáveis de classe `@@nome`, mas se o seu código-fonte contiver muitas variáveis de classe, há um sério problema de design.

Vejamos alguns exemplos para facilitar o entendimento:

▸ intro/32class.rb

```
class Carro
    def initialize(fabricante, modelo, ano)
        @fabricante = fabricante
        @modelo = modelo
        @ano = ano
    end
    attr_accessor :fabricante, :modelo, :ano
end
clio = Carro.new "Renault", "clio", "2000"
puts clio.modelo
```

Herança é um dos pilares da orientação a objetos (os outros dois são encapsulamento e polimorfismo). O Ruby suporta herança utilizando o operador < na definição de uma classe.

▶ intro/33inheritance.rb

```
require '32class'

class Clio < Carro
    @@fabricante = "Renault"
    @@modelo = "clio"

    def initialize(ano)
        super(@@fabricante, @@modelo, ano)
    end
end

clio = Clio.new(2003)
puts clio.modelo
```

Métodos de classe são bastante úteis no Ruby, pois podem saber a que classe pertencem. Para simplificar o exemplo, podemos também utilizá-los para criar uma fábrica de carros:

▶ intro/34classmethod.rb

```
require '33inheritance'

class Fabrica
    def self.clio
        Clio.new(2003)
    end

    def self.megane
        Carro.new "Renault", "megane", 2003
    end
end

puts Fabrica.clio.inspect
puts Fabrica.megane.inspect
```

Como foi possível perceber, para criar um método de classe é usada a notação `self.método`. É possível definir novos métodos em classes a qualquer momento, bem como definir novos métodos apenas em uma instância de uma classe, o que internamente vai fazer com que uma nova classe anônima seja criada apenas para aquele objeto.

Classes têm alguns métodos interessantes para entendermos o código Rails que será usado nos próximos capítulos. A seguir, veja uma breve lista.

Método	Descrição
`class`	Retorna a classe de um objeto.
`class_eval`	Executa uma string contendo código-fonte Ruby no contexto da classe.
`class_variable_defined?`	Informa se uma variável está definida nessa classe.
`class_variables`	Lista todas as variáveis de classe.
`const_defined?`	Informa se existe uma constante definida na classe.
`const_get`	Lê o valor de uma constante.
`const_set`	Grava o valor em uma constante ou cria uma nova.
`constants`	Lista todas as constantes definidas na classe.
`instance_eval`	Executa uma string contendo código-fonte Ruby no contexto de uma instância da classe.
`instance_methods`	Lista todos os métodos de instância da classe.
`instance_variable_defined?`	Informa se uma variável está definida para as instâncias da classe.
`instance_variable_get`	Lê o valor de uma variável de instância.
`instance_variable_set`	Cria ou altera o valor de uma variável de instância.

Métodos

Uma coisa interessante sobre métodos no Ruby é que eles não existem.

É exatamente isto: não há métodos em Ruby! A diferença é bastante sutil, mas ajuda a entender melhor como as coisas funcionam. No Ruby, não se chama um método de objeto; envia-se uma mensagem para um objeto, e essa pode ter parâmetros, mas sempre tem um retorno.

É como se cada objeto tivesse uma caixa de correio interna que só aceita mensagens para destinatários conhecidos e todos os destinatários desconhecidos fossem para o mesmo lugar: uma caixa com o nome de `method_missing`.

Isso permite que métodos inexistentes sejam adicionados apenas no momento em que se tornam necessários. Veremos isso acontecer bastante no código do Rails, cuja leitura recomendo após a conclusão deste livro, pois a melhor forma de entender profundamente como o Rails funciona é lendo o código-fonte do Rails.

Aqui está um exemplo de utilização do `method_missing`:

▸ intro/35method_missing.rb

```
class Teste
    def method_missing(method, *args)
        print "Método #{method} chamado na classe Teste, com os argumentos #{args.join(', ')}\n"
    end
end
t = Teste.new
t.imprimir
t.qualquer_coisa 1, 2, 3, "asd", :teste => 1
```

Não é considerada uma boa prática utilizar o `method_missing` o tempo todo, principalmente porque, no exemplo mostrado, se fosse feita a pergunta `t.respond_to? :imprimir`, o objeto diria que não responde à mensagem, mas ela seria enviada. Tudo funciona sem problemas, o que cria um objeto com comportamento inconsistente.

Uma melhor abordagem seria a utilização do `define_method` para criar uma caixa de correspondências, no momento em que esta se tornar necessária, da forma como o `attr_accessor` faz. Para demonstrar isso, criaremos um método que define propriedades em um objeto, semelhante ao `attr_accessor`. Para isso, vamos utilizar o módulo que é assunto da próxima seção.

▸ intro/36define_method.rb

```
module Propriedades
    def propriedade(nome)
        ivarname = "@#{nome}".to_sym
        self.send :define_method, nome do
            self.instance_variable_get ivarname
        end
        self.send :define_method, "#{nome}=".to_sym do |val|
            self.instance_variable_set ivarname, val
        end
    end
end

class Teste
    extend Propriedades
    propriedade :nome
end
```

```
t = Teste.new
t.nome = "urubatan"
print t.inspect
print "\n"
```

Módulos

Módulos no Ruby são repositórios de coisas. Essa foi a melhor explicação de módulos que consegui, falando de uma forma bem genérica.

Eles podem conter classes, sendo usados como pacotes de classes para organizar um domínio muito grande. Ou podem conter métodos para serem utilizados como "mixins", um conceito bastante interessante que, utilizado junto com as classes abertas, é parcialmente responsável por toda a flexibilidade do Ruby.

Para utilizar módulos como organizadores de classes, basta fazer como neste exemplo:

▸ intro/37modules.rb

```
module Administracao
    class Cliente
        attr_accessor :nome

        def initialize
            @nome = ""
        end
    end
end
```

Utilizar métodos como mixins é bastante semelhante ao exemplo anterior, mas dentro do módulo são definidos métodos, e não classes, como no exemplo a seguir.

▸ intro/38modulemixin.rb

```
class Teste
    def ola_mundo
        print "ola mundo\n"
    end
```

```
    def self.ola_mundo
        print "ola mundo da classe\n"
    end
end
Teste.ola_mundo
t = Teste.new
t.ola_mundo
module MixinTest
    def self.included(base)
        base.send :include, InstanceMethods
        base.send :extend, ClassMethods
    end

    module ClassMethods
        def metodo_de_classe
            print "Novo método de classe definido no módulo 'ClassMethods'\n"
        end
    end

    module InstanceMethods
        def metodo_de_instancia
            print "Novo método de instância definido no módulo 'InstanceMethods'\n"
        end
    end
end

class Teste
    include MixinTest
end

Teste.metodo_de_classe
t.metodo_de_instancia
```

Em qualquer classe, pode-se chamar o método `include` passando um módulo como parâmetro, e os métodos desse módulo estarão disponíveis para todas as instâncias dessa classe.

Se o método `extend` for utilizado, os métodos do módulo estarão disponíveis para a classe, e não para suas instâncias.

O método `send` utilizado no exemplo envia uma mensagem para um objeto. No caso, o objeto era a classe `Teste`.

Operadores condicionais e loops

Ruby é uma linguagem dinâmica, mas também imperativa, e todas as linguagens imperativas têm estruturas de controle de fluxo e loop. Nas próximas seções veremos quais estruturas desse tipo o Ruby contém e como utilizá-las.

Operadores condicionais

If / elsif / else / end

A estrutura `if` é utilizada para executar um conjunto de instruções. Se a condição for verdadeira, não é necessário o comando `then`, e o `if` pode ser utilizado no final de uma instrução também. Dessa forma, o bloco anterior de código só será executado se a condição for verdadeira.

▸ intro/39ifelse.rb

```
a = 0
if a==0
    print "zero"
elsif a==1
    print "um"
else
    print "não sei que número é este"
end
b = 5 if a!=1
puts b
```

unless else end

O comando `unless` é um atalho, mais fácil de ler para um "if not" em inglês. Ele facilita a leitura do código quando utilizado corretamente, mas semanticamente é um "if not" e pode ser usado também no final de uma sentença da mesma forma que o `if`.

▸ intro/40unlesslseend.rb

```
a = 1
unless a==0
    print "não é zero\n"
else
```

```
    print "a é zero\n"
end
b = 6 unless b
puts b
b = 7 unless b
puts b
```

Como pode ser visto no exemplo, o `unless` pode ser usado para definir o valor de uma variável apenas se ela ainda não tiver um valor. É necessário apenas tomar cuidado com essa abordagem se o valor esperado for um valor booleano.

case / when / else / end

O comando `case` é um atalho mais organizado e semântico para uma sequência de `elsif`. O comando `case` do Ruby é bastante flexível, mais do que em Java ou C++, por exemplo. Isso porque no Ruby ele pode ser utilizado com qualquer tipo de objeto e não apenas com números. Só não é possível misturar objetos, como pode ser visto no exemplo a seguir.

▶ intro/41casewhenelse.rb

```
a = 5
case a
    when 1..3
        puts "a entre 1 e 3\n"
    when 4
        puts "a=4\n"
    else
        puts "nenhuma das anteriores\n"
end
a = "b"
case a
    when "a"
        puts "a\n"
    when "b"
        puts "b"
    else
        puts "outra letra"
end
```

Operadores de loop

Os operadores apresentados nesta seção podem ser utilizados com qualquer um dos loops que serão apresentados nas próximas seções:

Operador	Descrição
`break`	Sai do loop atual.
`next`	Executa o próximo passo do loop.
`return`	Sai do laço e do método atual.
`redo`	Reinicia o loop atual.

while

O `while` é um loop bastante flexível, pois permite o controle da condição do laço, podendo ser utilizado com qualquer condição booleana, derivada da comparação de qualquer tipo de objeto.

▸ intro/42while.rb

```
i = %w{a b c d e f}
while b = i.pop
    puts b
end
```

No exemplo, o `while` foi utilizado para iterar sobre um array de strings. Pense nisso apenas como um exemplo, pois nesse caso o mais indicado seria utilizar o método `each` da classe `Array`.

for

O laço `for` é usado para repetir um bloco de código por um número conhecido de vezes. Utilize-o apenas quando for realmente necessário, pois o modo padrão do Ruby de iterar sobre coleções é empregando os métodos apropriados, como o `each`, por exemplo.

▸ intro/43for.rb

```
for i in 1..5
    puts i
end
for a in %w{a b c d}
    puts a
end
```

until

O bloco `until` é o contrário do `while`: ele repete o bloco de código até que a condição seja verdadeira.

▸ intro/44until.rb

```
i = 5
until i == 0
    puts i
    i -= 1
end
```

begin

O bloco `begin` é utilizado em conjunto com o `while` ou `until` quando se deseja que o bloco seja executado pelo menos uma vez. Assim, a condição fica no final do bloco, e não no início, como nos exemplos anteriores.

▸ intro/45begin.rb

```
i =- 5
begin
    puts i
    i += 1
end while i < 0
```

Nesse exemplo, se um `while` padrão fosse utilizado, o bloco não teria sido executado nenhuma vez.

loop

O laço `loop` é o laço mais flexível. Ele será executado até que encontre um comando `break` ou `return` no bloco.

▸ intro/46loop.rb

```
loop do
    puts "a"
    break if true
end
```

Padrões importantes

É muito importante, quando se aprende uma linguagem nova, não tentar programar na linguagem anterior com a sintaxe da linguagem nova. Para isso, é importante aprender alguns padrões bastante utilizados no Ruby. Há uma pequena lista de coisas que devem ser lembradas a seguir.

Nomes de arquivos

Nomes de arquivos utilizam letras minúsculas e sublinhado para separar palavras. Um arquivo .rb que contém a definição de uma classe de nome `ClienteEspecial` terá o nome `cliente_especial.rb`.

Um módulo segue o padrão de nomenclatura de classes e, na maioria dos casos, também define a estrutura de diretórios. Seguindo o exemplo, se a classe `ClienteEspecial` estiver definida dentro do módulo `Clientes`, o arquivo `cliente_especial.rb` estará dentro do diretório clientes.

O Ruby não impõe esses padrões. É possível definir todas as classes da aplicação no mesmo arquivo se desejado, mas a maior parte das aplicações Ruby segue padrões parecidos com os descritos.

Classes, atributos e métodos de acesso

Não é obrigatório o nome de uma classe começar com uma letra maiúscula, mas é bastante recomendado, pois dessa forma também se define uma constante que apontará para a classe, facilitando seu uso.

Algumas vezes, será necessário criar métodos de acesso para variáveis de instância da classe. O próprio `attr_accessor` faz exatamente isso.

Métodos de leitura de uma variável de instância têm o mesmo nome da variável, sem o caractere `@` no início, e métodos de escrita têm o mesmo nome terminado em `=`.

▸ intro/47standards.rb

```
class Teste
    attr_accessor :nome
end

t = Teste.new
puts t.methods.sort
```

Nomenclatura de métodos

Nomes de métodos no Ruby têm sempre todas as letras minúsculas e utilizam sublinhado _ para separar palavras.

Métodos que transformam um objeto em outro têm o nome iniciado por `to_`, como, por exemplo:

Método	Descrição
`to_s`	Transforma em `String`.
`to_i`	Transforma em `Fixnum`.
`to_a`	Transforma em `Array`.
`to_sym`	Transforma em um símbolo.

Lembre-se disso ao criar esse tipo de método.

Entrevista com David Heinemeier Hansson

David Heinemeier Hansson é o criador do Ruby on Rails e sócio da 37signals, uma das empresas mais "cool" atualmente, além de ser um "gentleman racer" e escritor, sendo coautor de pelo menos três livros.

Site: `http://loudthinking.com`

Qual foi a sua primeira linguagem de programação?

> Eu brinquei um pouco com Basic quando criança, mas nunca me animei muito. Acho que a primeira vez que fiz algo mais real foi com Microsoft ASP, mas essa fase passou rápido e eu comecei a usar PHP.

Quando começou a trabalhar no Basecamp, você já estava pensando em criar o Rails? Quero dizer, você já pensava em separar a infraestrutura e criar um framework de desenvolvimento, ou você decidiu por isso quando o produto já estava pronto?

> Não era um requisito desde o início, mas lá pela metade do projeto eu percebi que havia criado muitas ferramentas que outros poderiam achar bastante úteis. Nesse ponto, eu comecei a pensar em liberar isso para o público.

Jason Fried pensou em implementar o Basecamp em PHP. O que o levou a escolher Ruby em vez de PHP?

> Eu estava cansado de PHP. Achava a linguagem feia e muitas das abstrações que eu queria extrair do meu trabalho eram difíceis de representar em PHP. Estava na hora de passar para outra. O Ruby, por outro lado, tinha toda a agilidade do PHP, mas também tinha a beleza, coerência de design e muitas outras técnicas modernas de desenvolvimento. Ele apenas apareceu na hora certa.

A maioria dos profissionais que eu conheço que trabalham com Ruby on Rails também utiliza metodologias ágeis. Você acha que o Rails leva as pessoas a utilizar metodologias ágeis? Acha que as metodologias ágeis levam ao Rails? Acha que a forma pela qual você desenvolveu o Basecamp e o Rails tem algo a ver com isso?

> Com certeza. Quando você está utilizando um ambiente de desenvolvimento burocrático e prolífico, é natural procurar processos de desenvolvimento também burocráticos. Mas o Ruby por sí só é tão leve, flexível e ágil que qualquer burocracia desnecessária aparece instantaneamente como um obstáculo a ser removido.

Você tem alguma dica para aqueles que estão começando a programar hoje e estão pensando em estudar Ruby on Rails? Você acha que eles devem aprender primeiro Ruby e depois passar para o Rails ou podem aprender as duas coisas ao mesmo tempo?

> Eu acredito que eles devam aprender a programar com um objetivo. Programar por programar nunca me chamou atenção. Aprenda a programar porque você quer construir alguma coisa real. Então, se você pode construir algo interessante só com Ruby, tudo bem. Mas se você quer fazer uma aplicação web, deve começar aprendendo Ruby e Rails ao mesmo tempo.

Você fala bastante da forma correta de se fazer negócio, e você é coautor de dois livros sobre o assunto (*Getting Real* e *Rework*). Você acha que todo programador deveria ter esse assunto em mente? Ou esses livros são para um público diferente?

> A maior parte dos programadores trabalha para organizações comerciais ou tem seus próprios negócios. Saber o valor do negócio e da produtividade é extremamente recompensador. Pensar em você como

> meramente um programador que traduz, mas não questiona, os "requisitos" de negócio é uma definição muito fraca. Pensar em você mesmo como alguém que está apenas usando código para entregar os recursos mais valiosos e as correções mais importantes para o negócio é o tipo de pensamento dos vencedores.

Você tem alguma dica para aqueles que estão começando a programar e pensando em buscar o primeiro emprego com Ruby on Rails?

> Prove seu valor na comunidade open source. Existem muitos projetos em que você pode ajudar. Comece corrigindo bugs, escrevendo documentação e evolua daí. Na 37signals, a primeira coisa que olhamos dos desenvolvedores é a sua contribuição open source.

Na sua opinião: qual a importância de trabalhar em projetos open source para a carreira de um desenvolvedor?

> É importantíssimo se você quer trabalhar para a nova classe de empresas. Mas se tudo o que você quer é um cubículo em uma empresa de seguros, então provavelmente você vai se dar melhor fazendo certificações da Microsoft, mas se você quer um trabalho em uma empresa nova e atual, open source é importantíssimo.

Falando de linguagens de programação, você acha que todo programador deveria aprender mais de uma linguagem? É possível ser um bom programador conhecendo apenas uma?

> É muito bom conhecer o que existe por aí, é importante absorver impulsos de ângulos diferentes. Mas eu acho que "aprender uma linguagem de programação por ano" é exagerado. Eu não aprendi realmente uma nova linguagem de programação desde que comecei com Ruby, há seis anos.

E, para finalizar, você gostaria de dizer alguma outra coisa para os leitores deste livro?

> Construam algo útil. Toda a habilidade em desenvolvimento do mundo não vai te ajudar a deixar uma marca no universo se você não tiver a iniciativa de utilizar essa habilidade em algo significativo.