10 coisas que todo programador Java deve saber sobre Ruby

Balance On Rails
Fabio Akita
Original de Jim Weirich

Apresentação

- Jim Weirich
  - Consultor da Compuware
  - Programador Java
  - Entusiasta Ruby
- Fabio Akita
  - Consultor da Balance Consulting
  - Integrador SAP
  - Programador Java
  - Entusiasta Ruby on Rails
  - Autor de “Repensando a Web com Rails”

Começando do Começo

- Quando ensinava C para funcionários de uma grande empresa, era fácil saber qual linguagem os estudantes usavam olhando o estilo do código C que faziam.

- Certamente "*você pode escrever Fortran em qualquer linguagem*".

- Programadores Java investigando Ruby encontrarão muitas semelhanças.
- Existem classes e módulos, namespaces e escopos, variáveis de instância e métodos.
- Um programador Java se sentirá em cada nessa linguagem orientada a objetos.

- Portanto a tentação será de continuar programando em estilo Java.
- Claro, algumas coisas serão diferentes (a falta de declaração de tipos será o primeiro choque).
- Mas nada que não possa ser trabalhado com um pouco de esforço, ou perderemos uma oportunidade de ouro.

- "*Uma linguagem que não afeta seu jeito de pensar sobre programação não vale a pena aprender*"

  - Alan Perlis

The Ruby Way

- Isto não é mais um artigo do tipo "Ruby é melhor que Java".
- Esta apresentação visa ajudar programadores Java interessados a evitar o "escrever Java em Ruby" e aprender o "Ruby Way".

#10 aprenda convenções Ruby

- `NomesDeClasse`
- `nomes_de_metodos` e `nomes_de_variaveis`
- `metodos_fazendo_pergunta?`
- `metodos_perigosos!`
- `@variaveis_de_instancia`
- `$variaveis_globais`
- `ALGUMAS_CONSTANTES` ou `OutrasConstantes`

- Algumas convenções são reforçadas pela linguagem, outras são padrões usadas pela comunidade.

#9 tudo é um objeto

- Tudo que puder ser ligado a um nome de variável é um objeto completo.
- Isso tem consequências interessantes.

Classes são Objetos!

- Array é um nome constante ligado a um objeto de classe Array.
- Criar novos objetos não exigem sintaxe especial. Apenas enviamos “new” para o objeto de classe.
- `a = Array.new`

Fábricas são triviais

- Como Classes criam instâncias de si mesmos, são objetos ideais de fábrica (design pattern de Factory).

```
def criar_pela_fabrica(fabrica)
   fabrica.new
end

obj = criar_pela_fabrica(Array)
```

Sem primitivas!

- Até inteiros são objetos.

```
0.zero?     # => true
1.zero?     # => false
1.abs       # => 1
-1.abs      # => 1
1.methods   # => lista de métodos do objeto 1
2.+(3)      # => 5  (mesmo que 2+3)
10.class    # => Fixnum
(10**100).class     # => Bignum
```

nil é um objeto!

- Java:
  - null significa “sem referência para objeto”
  - Chamar um método de null é um erro
- Ruby:
  - nil é um objeto normal
  - Você nunca terá um NullPointerException!

```
a = nil
a.nil?     # => true
a.methods  # => list of methods
a.abs      # => NoMethodError
```

Coisas que não são objetos

- Nomes de variáveis não são objetos
- Uma variável não pode ter referência a outra variável
  - Existem maneiras, mas ninguém realmente precisa disso.

Mais coisas que não são objetos

- Blocos não são objetos
  - Mas isso é uma distinção sem diferença
  - Quando precisar disso, será automaticamente convertido a um objeto `Proc`

```
def com_bloco
  yield
end

com_bloco {
  # Nunca convertido
}
```

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

com_proc {
  # Convertido internamente
}
```

#8 (Quase) tudo é uma mensagem

- Toda a computação de Ruby acontece através de:
  - Ligação de nomes a objetos (atribuição)
  - Estruturas primitivas de controle (ex. if/else,while) e operadores (ex. defined?)
  - Enviando mensagens a objetos

Sim, tudo isso são mensagens ...

`string.index("x")`
- Envia: **`index`** (com argumento `"x"`)

`string.length`
- Envia: **`length`** (sem argumentos)

`run_status_reports`
- Envia: **`run_status_reports`** (para **`self`**, ele mesmo)

`1 + 2`
- Envia: **`+`** (com argumento **`2`**) ao objeto **`1`**

`array[i]`
- Envia: **`[]`** (com argumento **`i`**) para o **`array`**

Mensagens, não chamadas de função

- Programadores Java tendem a pensar em `obj.metodo()` como uma procura e chamada a uma função-membro.
- Programadores Ruby tendem a pensar em `obj.metodo` como um envio de uma mensagem a um objeto.

Qual a diferença?

- A diferença é sutil mas importante.

Que *tipo* de diferença?

- Considere a seguinte classe, ela consegue registrar todas as mensagens enviadas a ela e depois reproduzí-las a outro objeto.

```
class Gravador
  def initialize
    @mensagens = []
  end
  def method_missing(metodo, *args, &bloco)
    @mensagens << [metodo, args, bloco]
  end
  def reproduza_a(obj)
    @mensagens.each do |metodo, args, bloco|
      obj.send(metodo, *args, &bloco)
    end
  end
end
```

Reproduzindo ...

```ruby
require 'src/vcr'

vcr = Gravador.new
vcr.sub!(/Java/) { "Ruby" }
vcr.upcase!
vcr[11,5] = "Universe"
vcr << "!"

string = "Hello Java World"
puts string

vcr.reproduza_a(string)
puts string
```

- Saída

```
Hello Java World
Hello RUBY Universe!
```

Oportunidades de Mensagens

- Proxys remotos
  - Automaticamente redirecione qualquer mensagens a um objeto remoto
- Carregadores automáticos
  - Adaptador que escuta mensagens para um objeto e só instancia quando receber essa mensagem, agindo com um proxy. Ótimo para auto carregamento de objetos de banco de dados
- Decoradores
  - Intercepta as mensagens que quiser e passa o resto junto
- Objetos Falsos (Mock)
  - Apenas escreva os métodos que precisam ser falsos. Faça um proxy ou ignore os outros.
- Construtores (Builders)
  - Gere XML/HTML/Qualquer-coisa baseado em chamadas de métodos no construtor.

#7 Ruby é *Bem* mais dinâmico do que você espera

- Um dos grandes atrativos de Java sobre C++ eram as funcionalidades dinâmicas da linguagem.

- Você podia facilmente carregar classes em tempo de execução (run time), perguntar a um objeto sobre sua classe e métodos e até mesmo chamar métodos descobertos em run time.

Dinâmico além de Java

- Ruby leva o comportamento dinâmico muitos passos além de Java:

- `method_missing`
- Reflexão Fácil
- Classes Abertas
- Objetos Singleton
- Ganchos (Hooks) de definição
- Code Evaluation

Reflexão Fácil: Criar objeto

```
import java.lang.reflect.*;

public class Saudacao {

  private String valor;

  public Saudacao(String valor) {
      this.valor = valor;
  }

  public void mostrar() {
      System.out.println(valor);
  }
```

```
  public static Object criar(Class c,
    String valor)
     throws Exception {
     Constructor ctor =
       c.getConstructor(
         new Class[] { String.class }
       );
     return ctor.newInstance(
       new Object[] { valor } );
   }

   public static void main (String
    args[])
     throws Exception {
     Saudacao g = (Saudacao)
       criar(Saudacao.class,
       "Hello");
     g.mostrar();
   }
}
```

Reflexão Fácil: versão Ruby

```ruby
class Saudacao
  def initialize(valor)
    @valor = valor
  end
  def mostrar
    puts @valor
  end
end
```

```ruby
def criar(klass, valor)
  klass.new(valor)
end

g = criar(Saudacao,
  "Hello")
g.mostrar
```

Classes Abertas

- Métodos podem ser adicionados a classes a qualquer momento … mesmo classes padrão

```
class Integer
  def par?
    (self % 2) == 0
  end
end

p (1..10).select { |n| n.par? }
  # => [2, 4, 6, 8, 10]
```

- Esta funcionalidade é muito útil mas deve ser usada com cuidado

Métodos Singleton

- Métodos Singleton são definidos em objetos individuais, não classes

```
class Cachorro
end

rover = Cachorro.new
fido = Cachorro.new

def rover.fale
  puts "Rover Vermelho"
end

rover.fale  # => "Rover Vermelho"
fido.fale   # => NoMethodError
```

Ganchos (Hooks)

- Ganchos permitem o usuário ganhar controle em momentos interessantes durante a execução do programa

```
class MinhaClasse
  def MinhaClasse.method_added(nome)
    puts "Adicionado método #{nome}"
  end

  def novo_metodo
    # bla bla bla
  end
end
```

- Saída

```
Adicionado método novo_metodo
```

Code Evaluation

```ruby
class Module
  def trace_attr(sym)
    self.module_eval %{
      def #{sym}
        printf "Acessando %s com valor %s\n",
          "#{sym}", @#{sym}.inspect @#{sym}
      end
    }
  end
end
class Cachorro
  trace_attr :nome
  def initialize(string)
    @nome = string
  end
end
Cachorro.new("Fido").nome # => Acessando nome com valor"Fido"
```

#6 Objetos são Fortemente e não Estaticamente Tipadas

- ***O que é um Tipo?***

- Um Tipo é
  - **Um conjunto de *valores***

- E
  - **Um conjunto de *operações***

Código C (Fraco)

```c
#include <stdio.h>
extern float dois();
int main() {
    float x = 1.5 + dois();
    printf("%f\n", x);
    printf("%d\n", x);
    return 0;
}

int dois() { return 2; }
```

Saída

```
nan
0
```

Código Java (Forte)

```java
public class Main {
  public static
    void main (String args[]) {
    double x = 1.5 + Dois.dois();
    System.out.println(x);
  }
}
```

```java
public class Dois {
  public static int dois() {
    return 2;
  }
}
```

Saída

```
3.5
```

Código Ruby (?)

```
require ‘dois'

x = 1.5 + dois
puts x
printf "%d", x

def dois
  2
end
```

Saída

```
3.5
3
```

- Então o que faz uma linguagem ter tipagem segura (type safe)?
  - Conhecimento do compilador do tipo da variável?
  - Declarar todas as variáveis?
  - Compilador pegando todos os erros de tipo?
- Ou …
  - Pegar todas as operações não apropriadas para um tipo, seja em
    - tempo de compilação ou em
    - tempo de execução

Código Ruby

```ruby
def fatorial(n)
  resultado = 1
  (2..n).each do |i|
    resultado *= i
  end
  resultado
end



puts fatorial(20)
puts fatorial(21)
```

Saída

```
2432902008176640000
51090942171709440000
```

Código Java

```java
public class Fact {
  static long fatorial(long n) {
    long resultado = 1;
    for (long i=2; i<=n; i++)
      resultado *= i;
    return resultado;
  }

  public static
    void main (String args[]) {
    System.out.println(fatorial(20));
    System.out.println(fatorial(21));
  }
}
```

Saída

```
2432902008176640000
-4249290049419214848 // => erro
```

Sistemas de Tipos de Linguagem

- Java tem tipos
  - Fortes
  - Estáticos
  - Manifestados
- Ruby tem tipos
  - Fortes
  - Dinâmicos
  - Implícitos

Testemunho

- Bob Martin:
- Tenho sido defensor de tipos estáticos há anos. Aprendi da maneira difícil usando C. Sistemas demais caíram em produção por erros idiotas de tipos.
- Quatro anos atrás me envolvi com Extreme Programming. Não consigo imaginar não ter uma suíte de testes completa para garantir meu desenvolvimento.

Testemunho

- Bob Martin:

- Há dois anos notei uma coisa. Estava dependendo menos e menos do sistema de tipagem para garantir contra erros. Meus testes unitários estavam me impedindo de cometer erros de tipagem.

- Então tentei escrever alguns aplicativos em Python e então em Ruby. Não fiquei completamente surpreso quando vi que problemas com tipagem nunca aconteceram.

#5 Não se preocupe com Interfaces

- Ruby usa *Duck Typing*
  - Se anda como um pato
  - Se fala como um pato
  - Então o trataremos como um pato
  - (quem se interessa o que realmente é?)
- Não há necessidade de herdar de uma interface comum

```
class Pato
  def fale() puts "Quack" end
end
class ObjetoParecidoComPato
  def fale() puts "Kwak" end
end
grupo = [
  Pato.new,
  ObjetoParecidoComPato.new ]
for ave in grupo do
    ave.fale
end
```

Saída

```
Quack
Kwak
```

#4 Mixture com Mixins

- Embora Ruby não tenha interfaces, tem Mix ins definidos por Módulos
- Um Módulo ...
  - É um namespace (como uma classe)
  - Pode ter métodos definidos (como uma classe)
  - Não pode ser instanciado (diferente de uma classe)
  - Pode ser misturado (incluído) em uma classe
    - Os métodos do módulo se tornam métodos de instância da classe

Operadores de Comparação Tediosos

- Embora toda a lógica esteja definida no método menor-que, todas as outras comparações ainda precisam ser definidas

```
class Par
  attr_accessor :primeiro, :segundo
  # ...

  def <(outro)
    (primeiro < outro.primeiro) ||
      (primeiro == outro.primeiro && segundo < outro.segundo)
  end
  def >(outro)
    outro < self
  end
  # Outros métodos definidos em termos do menor-que:
  #     <=, >=, ==
end
```

Reuse o Mix in

- Um Mix in permite as partes comuns de serem reusadas

```
module ComparavelUsandoMenor
  def >(outro)
    outro < self
  end
  # Outros métodos definidos em termos de menor-que:
  #     <=, >=, ==
end

class Par
  include ComparavelUsandoMenor
  attr_accessor :primeiro, :segundo
  # ...
  def <(outro)
    (primeiro < outro.primeiro) ||
      (primeiro == outro.primeiro && segundo < outro.segundo)
  end
end
```

#3 Abrace Fechamentos

- Iteração

```
[1,2,3].each { |item| puts item }
```

- Gerenciamento de Recursos

```
conteudo = open(nome_arq) { |f| f.read }
```

- Eventos (Callbacks)

```
botao.on_click { puts “Botão foi apertado" }
```

Em vez disso …

```ruby
for item in @items
  puts @item
end
```

```ruby
f = File.open("test", "wb")
# faz alguma coisa com f
f.Close
```

Temos isso …

```ruby
@items.each { |i| puts i }
```

```ruby
File.open("test", "wb") do |f|
  # faz alguma coisa com f
end
```

Podemos usar blocos para criar DSLs (Domain Specific Language). Por exemplo, se quiséssemos redefinir nosso próprio "if"

```ruby
def meu_if(cond, &b)
  if cond
    b.call
  end
end
```

```ruby
a = 5
meu_if(a < 10) do
  puts "a é menor que 10"
  a += 1
end

# mostra "a é menor que 10"
# retorna 6
```

#2 ri é seu Amigo, irb é seu outro Amigo

- ri
  - Ruby Information. Listagem de instruções para objetos padrão do Ruby
- irb
  - Interactive Ruby. Interpretador Ruby na forma de um console

```
$ ri Array

---------------------------------------------------------- Module: Array
Arrays are ordered, integer-indexed collections of any object.
Array indexing starts at 0, as in C or Java. A negative index is
assumed to be relative to the end of the array---that is, an index
of -1 indicates the last element of the array, -2 is the next to
last element in the array, and so on.
------------------------------------------------------------------------
Includes:
---------
Enumerable(all?, any?, collect, detect, each_with_index, entries,
find, find_all, grep, include?, inject, map, max, member?, min,
partition, reject, select, sort, sort_by, to_a, zip)
Class methods:
-------------- [], new
Instance methods:
-----------------
&, *, +, -, <<, <=>, ==, [], []=, assoc, at, clear, collect,
collect!, compact, compact!, concat, delete, delete_at, delete_if,
each, each_index, empty?, eql?, fetch, fill, first, flatten,
flatten!, frozen?, hash, include?, index, indexes, indices, insert,
inspect, join, last, length, map, map!, nitems, pack, pop, push,
rassoc, reject, reject!, replace, reverse, reverse!, reverse_each,
rindex, select, shift, slice, slice!, sort, sort!, to_a, to_ary,
to_s, transpose, uniq, uniq!, unshift, values_at, zip, |
```

Outro exemplo de ri

- Pergunte sobre o método de instância `last` ...

```
$ ri Array#last

------------------------------------------------------------ Array#last
     array.last     =>  obj or nil
     array.last(n)  =>  an_array
------------------------------------------------------------------------
     Returns the last element(s) of _self_. If the array is empty, the
     first form returns +nil+.

        [ "w", "x", "y", "z" ].last   #=> "z"
```

Amostra de irb

- Adicione 1+2 e depois encontre os métodos definidos em `Proc` ...

```
$ irb --simple-prompt
>> 1 + 2
=> 3
>> Proc.instance_methods(false)
=> ["[]", "==", "dup", "call", "binding", "to_s",
    "clone", "to_proc", "arity"]
```

#1 Pare de escrever tanto código !

- Dito de colega de trabalho (parafraseado):
  - *Decidi tentar resolver meu problema em Ruby. Então escrevi um pouco de código e de repente descobri que havia terminado!*
- Exemplos:
  - Rake (Versão Ruby do Make)
  - Rails (Versão Ruby do conjunto de uma dezena de outros frameworks em qualquer outra linguagem)

#0 Ruby traz a Diversão de volta à Programação

Mais algumas coisas a saber

- Namespaces (Classes e Modules) são independentes de pacote

```
module ActiveRecord
  class Base
  end
end
```

- "." (ponto) VS "::" (duplo dois pontos)

```
r = ActiveRecord::Base.new
r.connection
```

- Interpolação de Strings

```
"Agora são " + Time.now.to_s(:short)
"Agora são #{Time.now.to_s(:short)}"
```

Mais algumas coisas a saber

- Não há Overloading em assinaturas de métodos
- Existe um projeto Jruby (Ruby for Java) (http://jruby.sourceforge.net)
- Existe um projeto IronRuby (Ruby for .NET)
- (http://wilcoding.xs4all.nl/Wilco/IronRuby.aspx)
- Método estático de Java e método de classe de Ruby são parecidos mas diferentes
- `finally` se chama `ensure`
- Aspas flexíveis:

```
"#{1 + 2}" # => "3"
'#{1 + 2}' # => "\#{1 + 2}"
```

Resumo

(10) Aprenda Convenções Ruby
(9) Tudo é um Objeto
(8) (Quase) tudo é uma Mensagem
(7) Ruby é Bem mais Dinâmico do que se espera
(6) Objetos são Fortemente e não Estaticamente Tipados
(5) Não se preocupe com Interfaces
(4) Mixture com Mixins
(3) Abrace Fechamentos
(2) ri é seu Amigo, irb é seu outro Amigo
(1) Escreva menos Código
(0) Ruby traz a Diversão de volta à Programação