

# [IF977] Engenharia de Software

---

Prof. Vinicius Cardoso Garcia  
[vcg@cin.ufpe.br](mailto:vcg@cin.ufpe.br) :: [@vinicius3w](https://twitter.com/vinicius3w) :: [assertlab.com](http://assertlab.com)



# Licença do material

---

Este Trabalho foi licenciado com uma Licença

**Creative Commons - Atribuição-NãoComercial-  
Compartilhagual 3.0 Não Adaptada.**

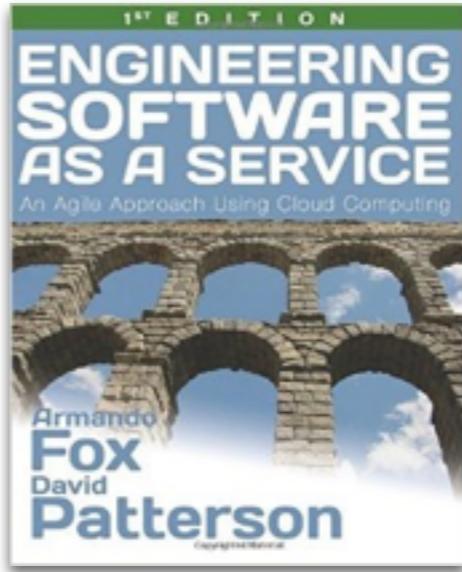
Mais informações visite

[http://creativecommons.org/licenses/by-nc-sa/3.0/  
deed.pt](http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt)



# Referências

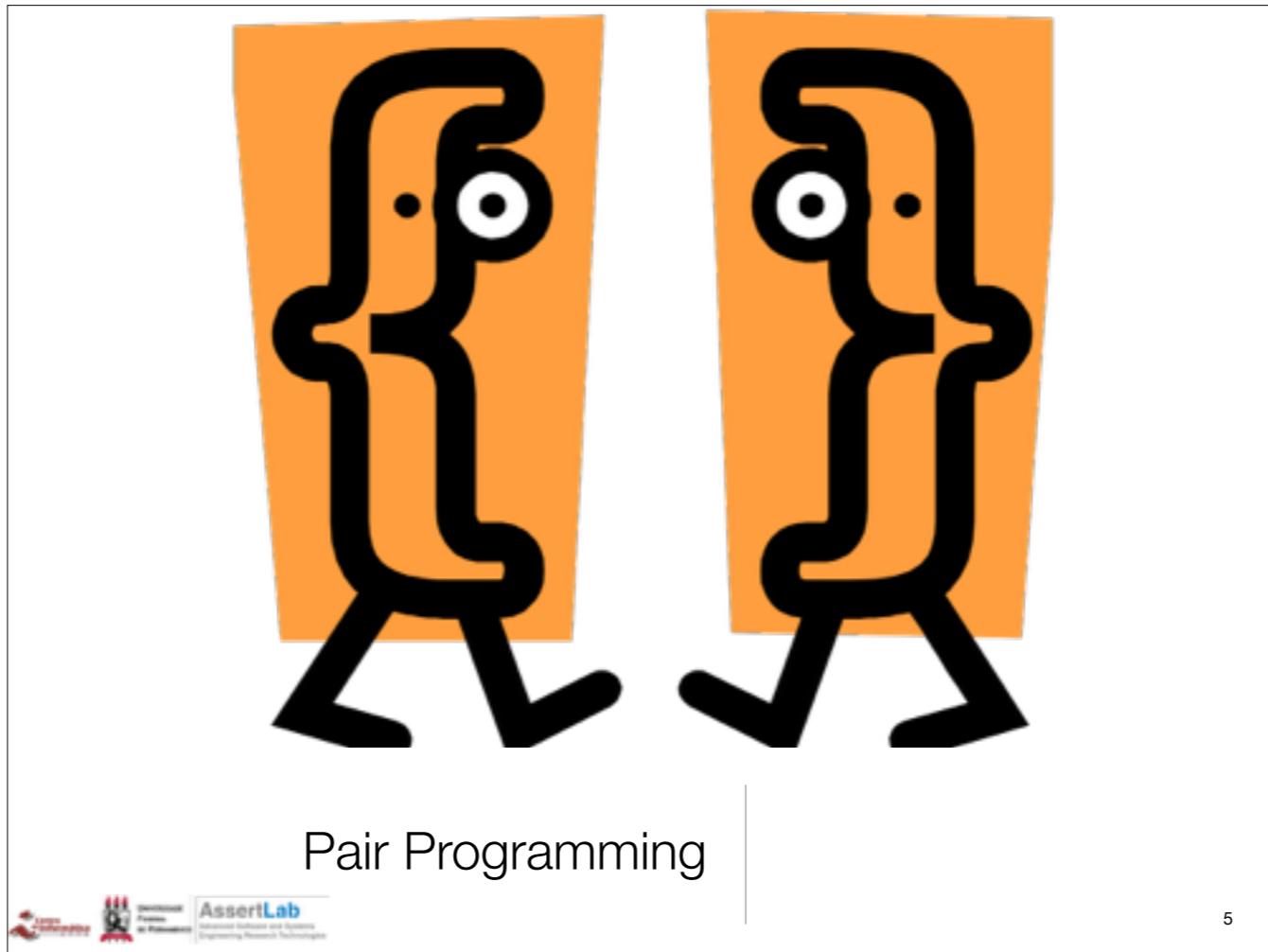
- A biblioteca do Desenvolvedor de Software dos dias de hoje
  - <http://bit.ly/TDOA5L>
- SWEBOK
  - Guide to the Software Engineering Body of Knowledge (SWEBOK): <http://www.computer.org/web/swebok>
- Engineering Software as a Service: An Agile Approach Using Cloud Computing (Beta Edition)
  - <http://www.saasbook.info/>



# Outline

---

- (ESaaS 10.2) Pair Programming
- (ESaaS 3.1) Overview & Three Pillars of Ruby
- (ESaaS 3.2-3.3) Everything is an object,  
Every operation is a method call
- (ESaaS 3.4) Ruby OOP (if time permits)

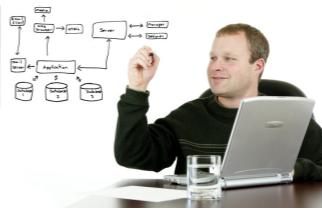


(Engineering Software as a Service §10.2)

University is a chance to make mistakes at university

## Duas mentes são melhores do que uma?

- O Estereótipo do programador é o lobo solitário que trabalha na madrugada Red Bull
- Há uma forma mais social de trabalhar?
  - Quais os benefícios de ter várias pessoas programando juntas?
  - Como prevenir que uma pessoa faça todo trabalho sozinha enquanto a outra está navegando no facebook?



# Programação em Pares

---

- Goal: **improve** software **quality**, **reduce time** to completion by having **2 people** develop the same code
  - Some people (and companies) **love it**
  - Try in discussion sections to see if you do
  - Some students in the past have loved it and used for projects

# Programação em Pares



- Sente-se lado a lado, com telas para ambos
  - Sem computador pessoal; muitos disponíveis para qualquer um usar
  - Para evitar distrações, sem leitor de email, browser, IM...

8

Sarah Mei and JR Boyens at Pivotal Labs

We know that MultiTasking is distracting, so these are real workstations with no facebook, email, twitter, ...

Don't check email between 8AM and 5PM

Productivity improves

New employees say tired after 8 hours

Go home at night and don't do programming at home

SV Management technique is buying dinner

# Programação em Pares

- **Guia** entra com o código e pensa estrategicamente como completar a tarefa, explicando as decisões enquanto codifica
- **Observador** revisa cada linha de código escrita, e atua como um dispositivo de segurança para o guia
- **Observador** pensa estrategicamente sobre o problemas e desafios futuros, e faz sugestões ao guia
- Deve haver **MUITA** conversa e concentração
- **Os papéis devem se alternar**

9

Talking out loud while work (like surgeon on TV)

Driver short term, Observer long term

Have to Alternative

# Avaliação da Programação em Pares

- PP acelera quando a complexidade da tarefa é pequena
- PP eleva a qualidade quando a complexidade é alta
  - Curiosamente, o código também fica mais legível
- Mais esforço do que na programação solo?
- Compartilhamento e conhecimento
  - Idiomas de programação, truques em ferramentas, processos, tecnologias...
  - Muitos times propõe trocar os pares por tarefa
    - Eventualmente, todos podem estar “**pareados**” (“*promiscuous pairing*”)

10

But Pivotal Labs denies it

More senior and more junior

“promiscuous pairing”

Does it factor in distraction

# PP: fazer & Não Fazer

- **Não** mexa no seu smartphone quando for o observador
- **Considere** formar o par com alguém com diferente nível de experiência que você – os dois irão aprender algo!
- **Forme** pares frequentemente – o aprendizado é bidirecional e papéis exercitam diferentes habilidades
  - O observador exerce a habilidade de explicar seu processo de pensamento ao guia

11

Concentrate on task at hand (not phone)

Great if mismatch to learn

# Pergunta

---

Qual expressão sobre Programação em Pares é **VERDADEIRA**?

- A. PP é mais rápida, barata e com menos esforço do que programação solo
- B. O guia trabalha nas tarefas na mão, o observador pensa estrategicamente sobre as tarefas futuras
- C. “*Promiscuous paring*” é uma solução a longo prazo para o problema da escassez de programadores
- D. O par vai, eventualmente, descobrir quem é melhor como guia e como observador e em seguida define quem fica em cada função

12

VertLeftWhiteCheck1

1 False: While quicker, While there have not been careful experiments that would satisfy human subject experts, the current consensus is that pair programming is more expensive—more programmer hours per tasks—than solo programming.

2. True

3. An effective pair will alternate between the two roles, as it's more beneficial (and more fun) for the individuals to both drive and observe

4. Trying to be funny

# Pergunta

---

Qual expressão sobre Programação em Pares é **VERDADEIRA**?

- A. PP é mais rápida, barata e com menos esforço do que programação solo
-  B. O guia trabalha nas tarefas na mão, o observador pensa estrategicamente sobre as tarefas futuras
- C. “*Promiscuous paring*” é uma solução a longo prazo para o problema da escassez de programadores
- D. O par vai, eventualmente, descobrir quem é melhor como guia e como observador e em seguida define quem fica em cada função

13

VertLeftWhiteCheck1

1 False: While quicker, While there have not been careful experiments that would satisfy human subject experts, the current consensus is that pair programming is more expensive—more programmer hours per tasks—than solo programming.

2. True

3. An effective pair will alternate between the two roles, as it's more beneficial (and more fun) for the individuals to both drive and observe

4. Trying to be funny

*“Rails is the killer app for Ruby.”*  
– Yukihiro Matsumoto, Criador da linguagem Ruby



## Overview & Intro to Ruby

for Java programmers

14



(Engineering Software as a Service §3.1)

# Ruby é...

- Interpretada
- Orientada a objetos
  - Tudo é um objeto
  - Toda operação é uma chamada de método em algum objeto
- Dinamicamente tipada: objetos possuem tipos, mas variáveis não
- Dinâmica
  - Adicione e modifique código em runtime (metaprogramação)
  - Pergunte aos objetos sobre eles mesmo (reflexão)
  - Filosoficamente, toda programação é metaprogramação

15

It is a small language, once you get a few ideas, much easier to understand

No compile time, no compile time errors

OO: A pillar of the language - Once understand that everything is an object

Only way to do anything is to send a message to an object, will save you grief in understanding what is going on

Dynamically typed: Can't tell type with certainty until runtime, but while objects have types, variables that refer to them don't have types  
(like Python unlike Java)

Highly dynamic – lots of things can do at runtime that are compile time in other languages

There is no other thing than runtime in Ruby; even class declarations are not code declarations, they are code that gets executed to create new things at run time  
Ruby feature makes code more concise (Productivity!) and more elegant

Reflection and metaprogramming often used together

it's always runtime, so ALL programming is metaprogramming – even loading a class causes code to be executed.

# Convenção de nomes

ClassNames usa UpperCamelCase

```
class FriendFinder ... end
```

methods & variables usa snake\_case

```
def learn_conventions ... end
```

```
def faculty_member? ... end
```

```
def charge_credit_card! ... end
```

CONSTANTS (scoped) & \$GLOBALS (not scoped)

```
TEST_MODE = true           $TEST_MODE = true
```

symbols: immutable string whose value is itself

```
favorite_framework = :rails
```

```
:rails.to_s == "rails"
```

```
"rails".to_sym == :rails
```

```
:rails == "rails" # => false
```

16

Every language has naming conventions

Camel case has humps in it (“Title Case”); UpperCamelCase starts with capital letter;  
snake\_case has little underscores between words

Define class use UpperCamelCase

method & variable names use snake\_case

Note: Some method names end optionally with a ? or with a ! (“bang”);

By convention, ? means method returns a Boolean

By convention, ! that means method does something destructive, or can't be undone

Convention, but in language

CONSTANTS in caps – like variables, only visible in scope in which defined

Also \$GLOBALS in red because dangerous; don't use them; begin with \$, uppercase

Immutable (can't change), and value is itself

Begin with :, don't have to be lowercase, but usually are

Not same as a string; can convert back and forth from string to symbol, but not equal

# Variáveis, Arrays e Hashes

- Não há declaração!
  - Variáveis locais devem ser atribuídas antes do seu uso
  - Variáveis de classe e de instância == nil até que sejam atribuídas
- Ok: `x = 3; x = 'foo'`
- **Errado:** `Integer x=3`
- Array:
  - `x = [1, 'two', :three]`
  - `x[1] == 'two'; x.length == 3`
- Hash:
  - `w = {'a' => 1, :b=>[2, 3]}`
  - `w[:b][0] == 2`
  - `w.keys == ['a', :b]`

17

Note, Array and hash elts can be anything, including other arrays/hashes, and don't all have to be same type. As in python and perl, hashes are the swiss army chainsaw of building data structures.

# Métodos

---

Tudo (exceto Fixnums) é passado por referência

```
def foo(x,y)
    return [x,y+1]
end

def foo(x,y=0) # y is optional, 0 if omitted
    [x,y+1]      # last exp returned as result
end

def foo(x,y=0) ; [x,y+1] ; end
```

Chame com a, b = foo(x,y)

Ou a, b = foo(x) quando o arg opcional for usado

# Construtores básicos

- Declarações terminadas em ';' ou nova linha, mas podem se estender por mais linhas se a análise não for ambígua

- ✓`raise("Boom!") unless (ship_stable)`      ✗ `raise("Boom!") unless (ship_stable)`

- Comparações básicas & booleans

- `== != < > =~ !~ true false nil`

- Controles de fluxo usuais

<code>if cond (or unless cond) statements [ elsif cond statements ] [ else statements ] end</code>	<code>while cond (or until cond) statements end 1.upto(10) do  i  ... end 10.times do...end collection.each do  elt ...end</code>
--	---

19

Breaking after unless OK, since it can't be end of statement

In contrast, on right not OK, because raise ("Boom!") is a legal statement, and parser will complain about unless

Show example might see in code

Basic

Boolean constants: true false and nil

False and nil in conditional statement evaluate to false; everything else is true

(Not like Python or PERL where empty string or empty array evaluate to false); only 2 false

Usual control flow

(Loops not common, just a special case of common Ruby construct called an iterator)

# Strings & Expressões regulares

```
"string", %Q{string}, 'string', %q{string}
a=41 ; "The answer is #{a+1}"
```

Encontrar a string em comparação a uma regexp:

```
"vcg@cin.ufpe.BR" =~ /(.*@\.(.*))\.br$/i
/(.*@\.(.*))\.br$/i =~ "vcg@cin.ufpe.BR"
```

Se não encontrar, o valor é false

Se encontrar, o valor é non-false, e `$1...$n` captura grupos entre parêntesis  
`($1 == 'vcg', $2 == 'ufpe')`

```
/(.*)$/i ou %r{(.*)$}i
ou Regexp.new('(.*)$', Regexp::IGNORECASE)
```

Um exemplo real...

<http://pastebin.com/hXk3JG8m>

20

Regexes in other languages comparable to Ruby

Lots of ways to represent strings

Double quotes mean can put expressions inside using #{} (see example)

String with single quotes have nothing done to them

%Q and %q – ways to represent strings with special characters in them (“quote them”). One way to put “ into strings.

How compare? Borrowed from Python: =~ operator

RHS regex first example matches an email address (not that it captures the substrings too)

Can also reverse sense of comparison

If not false in comparison, then capture expression groups in \$1, \$2, ... what is in ()s

To give you a chance to read code, once in a while we will flash code snippets so to acclimatize you to what this will look like in real life

# Pergunta

```
rx = { :vcg=>/^arm/,
       'vcg'=>[ %r{AN(DO)$}, /an(do) /
i ] }
```

Qual expressão vai ser avaliada como non-nil?

- A. “vinicius” =~ rx{:vcg}
- B. rx[:vcg][1] =~ “VINICIUS”
- C. rx['vcg'][1] =~ “VINICIUS”
- D. “vinicius” =~ rx['vcg', 1]

21

Hash, complicated expressions, which will evaluate to a successful match of a reg expression?

Correct answer is:

Why these wrong?

Orange: “vinicius” =~ rx{:vcg}

Wrong: Declare with braces {}, but use with square brackets [], so orange is out

Green: rx[:vcg][1] =~ “VINICIUS”

:vcg points to first elements of hash, but then applies [1] to get 1-th element of array, but not an array but a regex, so wrong

Red: rx['vcg'][1] =~ “VINICIUS”

‘vcg’ is element of hash, and its an array, so can take 2nd element of array since arrays are 0 based (as in all languages), so would match name (/I means ignore case), so it is correct

Blue: “vinicius” =~ rx['vcg', 1]

Looks like a multidimensional array, but really isn’t. it’s a hash. It is not a legal key.

Q: If it had been ‘vcg, 1’ it would have been legal key

# Pergunta

```
rx = { :vcg=>/^arm/,
       'vcg'=>[ %r{AN(DO)$}, /an(do) /
i ] }
```

Qual expressão vai ser avaliada como non-nil?

- A. "vinicius" =~ rx{:vcg}
- B. rx[:vcg][1] =~ "VINICIUS"
-  C. rx['vcg'][1] =~ "VINICIUS"
- D. "vinicius" =~ rx['vcg', 1]

22

Hash, complicated expressions, which will evaluate to a successful match of a reg expression?

Correct answer is:

Why these wrong?

Orange: "vinicius" =~ rx{:vcg}

Wrong: Declare with braces {}, but use with square brackets [], so orange is out

Green: rx[:vcg][1] =~ "VINICIUS"

:vcg points to first elements of hash, but then applies [1] to get 1-th element of array, but not an array but a regex, so wrong

Red: rx['vcg'][1] =~ "VINICIUS"

'vcg' is element of hash, and its an array, so can take 2nd element of array since arrays are 0 based (as in all languages), so would match name (/I means ignore case), so it is correct

Blue: "vinicius" =~ rx['vcg', 1]

Looks like a multidimensional array, but really isn't. It's a hash. It is not a legal key.

Q: If it had been 'vcg, 1' it would have been legal key



Tudo é um objeto, Toda operação é uma chamada de método

# Linguagens OO Modernas

---

- Objetos
- Atributos (propriedades), getters & setters
- Métodos
- Sobrecarga de operadores
- Interfaces
- “Boxing” e “unboxing” de tipos primitivos
- ... existe um pequeno conjunto de mecanismos que capturam a maioria destas características?

# Mundo OO

---

- Ruby é uma linguagem puramente orientada a objetos, bastante influenciada pelo Smalltalk.
- Desta forma, tudo em Ruby é um objeto, até mesmo os tipos básicos que vimos até agora.
- Uma maneira simples de visualizar isso é através da chamada de um método em qualquer um dos objetos:
  - `"strings são objetos".upcase()`
  - `:um_simbolo.object_id()`
- Até os números inteiros são objetos, da classe **Fixnum**:
  - `10.class()`

# Tudo é um objeto; (quase) tudo é uma chamada de método

- Mesmo inteiros insignificantes e nil são verdadeiros objetos:
  - `57.methods`
  - `57.heinz_varieties`
  - `nil.respond_to?(:to_s)`
- Reescrevendo cada uma delas como chamadas para o send:
  - Exemplo: `my_str.length => my_str.send(:length)`
    - `1 + 2` `1.send(:+, 2)`
    - `my_array[4]` `my_array.send(:[], 4)`
    - `my_array[3] = "foo"` `my_array.send(:[]=, 3, "foo")`
    - `if (x == 3) ....` `if (x.send(:==, 3)) ...`
    - `my_func(z)` `self.send(:my_func, z)`
  - Em particular, coisas como “conversão implícita” em comparações não estão no sistema de tipos, mas sim na instância de métodos

26

EVERYTHING, even lowliest integer, is an object, and all operations, even +, is a method call

`57.methods` lists all the methods that could work on this number

Even `nil` can answer questions about itself

Can `nil` respond to `to_s` (to string)? Answer is yes

Everything in the language is just syntactic sugar for a method call

Send means send this method call to this object

`my_str.length` really means

`my_str.send`

(send is method on every object)

Means send `my_str` the method name `:length`

All things on left that look like part of the Ruby language, really aren't part of the Ruby language

`1 + 2` is syntactic sugar for sending the `+` operator and the object `2` to the additional argument `1`

`my_array[4]` Is just sending the method `[:] (bracket name)` and the additional argument `4` to the object `my_array`

# Lembre-se!

---

- **a . b** significa: chame método **b** no objeto **a**
  - **a** is the *receiver* to which you *send* the method call, assuming **a** will *respond to* that method
- **não significa:** **b** é uma variável de instância de **a**
- **não significa:** **a** é um tipo de estrutura de dados que possui **b** como um membro

*Entendendo esta distinção vai te poupar de muito sofrimento e confusões*

## Exemplo: toda operação é uma chamada de método

---

- `y = [1,2]`
- `y = y + ["foo",:bar] # => [1,2,"foo",:bar]`
- `y << 5 # => [1,2,"foo",:bar,5]`
- `y << [6,7] # => [1,2,"foo",:bar,5,[6,7]]`
  
- “`<<`” **destrutivamente modifica** o receptor, o “`+`” não
  - Métodos destrutivos geralmente possuem nomes terminados em “`!`”
  
- **Lembre-se! Trata-se de quase todos métodos instanciados de Array – não são operadores da linguagem!**
- Então `5+3`, `"a"+"b"`, `[a,b]+[b,c]` são diferentes métodos chamados ‘`+`’
  - `Numeric#+`, `String#+`, `Array#+`, para ser específico

# Hashes & Poetry Mode

- `h = {"stupid" => 1, :example=> "foo" }`
- `h.has_key?("stupid") # => true`
- `h["not a key"] # => nil`
- `h.delete(:example) # => "foo"`
- Idioma Ruby: “**poetry mode**”
  - Usando hashes para passar argumentos “keyword-like”
  - Omite chaves quando o último argumento para função é um hash
  - Omite parênteses para argumentos de funções
  - `link_to("Edit",{:controller=>'students', :action=>'edit'})`
  - `link_to "Edit", :controller=>'students', :action=>'edit'`
  - Na dúvida, **PARENTESES NELE!**

29

There are cases when it is legal to omit:

Braces around hashes {}, and

Parentheses around arguments ()

When it is unambiguous

Most common case is when hash is last argument

2nd version: lack of parentheses really nice for reading code

# Hashes

---

```
config = Hash.new

    config[ "porta" ] = 80
    config[ "ssh" ] = false
    config[ "nome" ] = "cin.ufpe.br"

    puts config.size
    # => 3

    puts config[ "ssh" ]
    # => false
```

# Poetry mode em ação

```
a.should(be.send(:>=, 7))
```

```
a.should(be() >= 7)
```

```
a.should be >= 7
```

```
(redirect_to(login_page)) and return()  
unless logged_in?
```

```
redirect_to login_page and return unless  
logged_in?
```

31

Here is example of combining poetry mode and syntactic sugar

Beauty is not for computer, but for other humans to read and maintain code

1st expression; should is method call to object a  
be is a method call, but with two arguments be() and number 7

Could say just syntactic sugar, but could also say is pending 6 hours a day reading core, which would I rather read?  
We vote for latter option

Q: Why `:>=`, because send a method name, which is usually a symbol

Q: Why `()` after `be`? To make sure it's a method call

Q: What is `be` an instance method of? Instance of an “expectation matcher”

(Setting up condition, then run some code, and later trigger method to be evaluated, which `be` is an instance method of

When can read without `()`s, lots easier to read.

# Pergunta

```
def foo(arg,hash1,hash2)
```

```
  ...
```

```
end
```

O que não é uma chamada válida para `foo()`?

- A. `foo a, {:x=>1, :y=>2}, z=>3`
- B. `foo(a, :x=>1, :y=>2, :z=>3)`
- C. `foo(a, {:x=>1, :y=>2}, {:z=>3})`
- D. `foo(a, {:x=>1}, {:y=>2, :z=>3})`

32

## Syntactic sugar test

Method named `foo` that we are defining

1st argument – call it a string

2nd argument – hash

3rd argument – hash

Which is NOT legal way to call `foo`?

Orange: `foo a, {:x=>1,:y=>2}, :z=>3`

Green: `foo(a, :x=>1, :y=>2, :z=>3)`

NOT: Ambiguity, `foo` takes 2 hashes, but not clear which hash is which; that is, which pair vs. single

Red: `foo(a, {:x=>1,:y=>2},{:z=>3})`

Blue: `foo(a, {:x=>1}, {:y=>2,:z=>3})`

# Pergunta

```
def foo(arg,hash1,hash2)
```

```
  ...
```

```
end
```

O que não é uma chamada válida para `foo()`?

- A. `foo a, {:x=>1, :y=>2}, z=>3`
-  B. `foo(a, :x=>1, :y=>2, :z=>3)`
- C. `foo(a, {:x=>1, :y=>2}, {:z=>3})`
- D. `foo(a, {:x=>1}, {:y=>2, :z=>3})`

33

## Syntactic sugar test

Method named foo that we are defining

1st argument – call it a string

2nd argument – hash

3rd argument – hash

Which is NOT legal way to call foo?

Orange: `foo a, {:x=>1,:y=>2}, :z=>3`

Green: `foo(a, :x=>1, :y=>2, :z=>3)`

NOT: Ambiguity, foo takes 2 hashes, but not clear which hash is which; that is, which pair vs. single

Red: `foo(a, {:x=>1,:y=>2},{:z=>3})`

Blue: `foo(a, {:x=>1}, {:y=>2,:z=>3})`



## Programação Orientada a Objetos em Ruby

(Engineering Software as a Service §3.4)

# Classes

---

```
class Pessoa
  def fala
    puts "Sei Falar"
  end

  def troca(roupa, lugar="banheiro")
    "trocando de #{roupa} no #{lugar}"
  end
end
```

- O diferencial de classes em Ruby é que são abertas. Ou seja, qualquer classe pode ser alterada a qualquer momento na aplicação. Basta “reabrir” a classe e fazer as mudanças

# Self

---

- Um método pode invocar outro método do próprio objeto.
- Para isto, basta usar a referência especial **self**, que aponta para o próprio objeto. É análogo ao **this** de outras linguagens como Java e C#.
- **Todo** método em Ruby é **chamado** em algum objeto, ou seja, um método é **sempre** uma mensagem enviada a um objeto. Quando não especificado, o destino da mensagem é sempre **self**

```
class Conta
  def transfere_para(destino, quantia)
    debita quantia
    # mesmo que self.debita(quantia)

    destino.deposita quantia
  end
end
```

# Classes e Herança

```
class SavingsAccount < Account      # inheritance
  # constructor used when SavingsAccount.new(...) called
  def initialize(starting_balance=0) # optional argument
    @balance = starting_balance
  end
  def balance    # instance method
    @balance    # instance var: visible only to this object
  end
  def balance=(new_amount)  # note method name: like setter
    @balance = new_amount
  end
  def deposit(amount)
    @balance += amount
  end
  @@bank_name = "MyBank.com"      # class (static) variable
  # A class method
  def self.bank_name  # note difference in method def
    @@bank_name
  end
  # or: def SavingsAccount.bank_name ; @@bank_name ; end
end
```

<http://pastebin.com/m2d3myyP> 37

# Pergunta

---

Qual dessas está correta?

1. my\_account.@balance
2. my\_account.balance
3. my\_account.balance()?
  - A. Todas as alternativas
  - B. Somente a [2]
  - C. [1] e [2]
  - D. [2] e [3]

# Pergunta

---

Qual dessas está correta?

1. my\_account.@balance
  2. my\_account.balance
  3. my\_account.balance( )?
- A. Todas as alternativas
- B. Somente a [2]
- C. [1] e [2]
-  D. [2] e [3]

## Atributos e propriedades: acessores e modificadores

---

- Atributos, também conhecidos como **variáveis de instância**, em Ruby são sempre privados e começam com @.
- **Não há como alterá-los de fora da classe**; apenas os métodos de um objeto podem alterar os seus atributos (encapsulamento!).

# Variáveis de instância

---

```
class SavingsAccount < Account

    def initialize(starting_balance)

        @balance = starting_balance

    end

    def balance

        @balance

    end

    def balance=(new_amount)

        @balance = new_amount

    end

end
```

# Variáveis de instância

---

```
class SavingsAccount < Account

  def initialize(starting_balance)

    @balance = starting_balance

  end

  attr_accessor :balance

end
```

**attr\_accessor** é somente um velho método que utiliza metaprogramação... **não** é parte da linguagem

# Pergunta

---

```
class String
  def curvy?
    !( "AEFHJKLMNTVWXYZ".include?(self.upcase))
  end
end
```

- A. String.curvy?("foo")
- B. "foo".curvy?
- C. self.curvy?("foo")
- D. curvy?("foo")

# Pergunta

---

```
class String
  def curvy?
    !( "AEFHJKLMNTVWXYZ".include?(self.upcase))
  end
end
```

- A. String.curvy?("foo")
-  B. "foo".curvy?
- C. self.curvy?("foo")
- D. curvy?("foo")

# Características próprias de Ruby (até agora)

---

- Orientação a objetos sem herança múltipla
  - Tudo é um objeto, mesmo inteiros simples
  - Classes, variáveis de instância invisíveis fora da classe
- Tudo é uma chamada de método
  - Normalmente, só importa se o receptor responde a um método
  - Maioria dos operadores (+, ==) instancia métodos
  - Dinamicamente tipada: objetos possuem tipos; variáveis não
- Métodos destrutivos
  - Maioria dos métodos são não-destrutivos, retornando uma nova cópia
  - Exceções: <<, alguns métodos destrutivos (e.g., formato **merge** Vs. **merge!** para hash)
- Idiomaticamente, {} e () são algumas vezes opcionais

# Concluindo

---

- Ciclos de vida: Dirigido a Plano (cuidadosamente planejado e documentado) vs Ágil (aberto a mudanças)
- Times: Scrum como times auto-organizados + papéis de Scrum Master e Product Owner
- Programação em Pares: aumenta qualidade, reduz tempo, educa os colegas
- Gerente de Projeto DP é o chefe
- Ruby: OO sem herança múltipla, tudo é uma chamada de método, métodos não-destrutivos, às vezes () e {} opcional

# Desafio

---

**Resolução da Lista de Exercícios 0:  
Introdução a Ruby (envio até D+14,  
23:59)**

**<http://bit.ly/if977-hw0>**