

Linguagem Ruby



PUC Minas
Poços de Caldas

Luiz Alberto Ferreira Gomes

Curso de Ciência da Computação

7 de agosto de 2019

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos

Ruby

- Linguagem inventada por Yukihiro "Matz" Matsumoto
- Versão 1.0 liberada em 1996(Japão)
- Popularizada no início de 2005 pelo Rails



Ruby

- Linguagem **dinâmica** e **orientada a objetos**
- Elegante, **expressiva** e declarativa
- Influenciada pelo Perl, Smalltalk, Eiffel e Lisp

..Java..

```
1  public class Print3Times {  
2      public static void main(String[] args) {  
3          for(int i = 0; i < 3; i++) {  
4              System.out.println("Hello World!")  
5          }  
6      }  
7  }
```

..Ruby..

Listing 1: hello.rb

```
1 # um comentario em ruby
2 3.times { puts "Hello World" }
```

Básico do Ruby

- Indentação de **2 espaços** para cada nível aninhado (**recomendado**)
- `#` é utilizado para comentários
 - use com moderação, o código deve ser **auto documentado**
- Scripts utilizam a extensão `.rb`

Listing 2: hello.rb

```
1  # um comentario em ruby
2  3.times { puts "Hello World" }
```

Convenção de Nomes

■ Variáveis e Métodos

- em **minúsculas** e separada_por_sublinhado (tenha mais de uma palavra)
- métodos ainda permitem no final os caracteres ?!

■ Constantes

- tanto TODAS_AS_LETRAS_EM_MAIUSCULAS ou no formato CamelCase

■ Classes(e módulos)

- formato CamelCase

Remoção do Ponto-e-Vírgula

- Não coloque o ponto-e-vírgula no final da linha
- Pode ser utilizado para colocar várias declarações em uma linha
 - altamente desencorajado

```
1 a = 3
2 a = 3; b = 5
```

Interactive Ruby (IRB) (1)

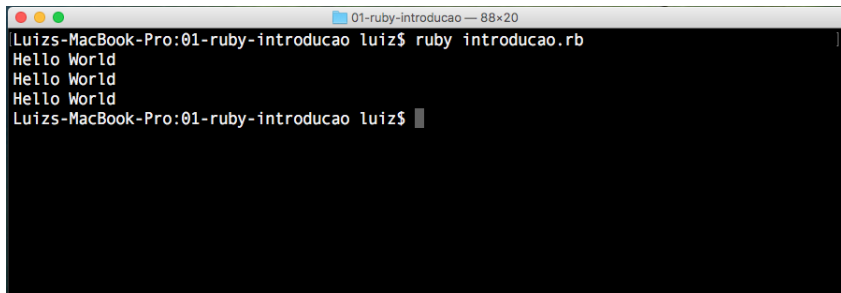
- Console **interativa** para interpretação de comandos Ruby
- Instalado com o interpretador Ruby
- Permite a **execução** de comandos rapidamente

A screenshot of a macOS terminal window titled "01-ruby-introducao — 88x20". The terminal shows a user at a "Luizs-MacBook-Pro" prompt running the "irb" command. Inside the IRB prompt, the user enters a Ruby command to print "Hello World!" three times. The output shows "Hello World!" printed three times, followed by the return value "3".

```
Luizs-MacBook-Pro:01-ruby-introducao luiz$ irb
irb(main):001:0> 3.times{ puts "Hello World!" }
Hello World!
Hello World!
Hello World!
=> 3
irb(main):002:0> 
```

Interactive Ruby (IRB) (2)

- Permite a **execução** de **scripts** contendo vários comandos

A screenshot of a macOS terminal window. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left and the text '01-ruby-introducao — 88x20' on the right. The terminal content shows a prompt 'Luizs-MacBook-Pro:01-ruby-introducao luiz\$' followed by the command 'ruby introducao.rb'. The output of the script is three lines of 'Hello World'. The prompt is then shown again with a cursor, indicating the command has finished.

```
Luizs-MacBook-Pro:01-ruby-introducao luiz$ ruby introducao.rb
Hello World
Hello World
Hello World
Luizs-MacBook-Pro:01-ruby-introducao luiz$
```

Ruby

- 1 Introdução
- 2 Entrada e Saída**
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos

Entrada pelo Teclado

- `gets` é método **padrão para receber** um valor pelo teclado

```
1 # recebe um valor do tipo string.  
2 nome = gets
```

- Utilize `gets.chomp` para remover o caracter de nova linha.

```
1 # remove o caracter de nova linha.  
2 nome = gets.chomp
```

- Utilize `gets.chomp.to_i` para converter o valor lido para inteiro.

```
1 # converte a string recebida para inteiro.  
2 idade = gets.chomp.to_i
```

Saída na Tela

- puts é método **padrão** para impressão em tela
 - insere uma quebra de linha após a impressão
 - similar ao `System.out.println` do Java

```
1  # exibe da tela do computador.  
2  puts "Informacoes do jogador"  
3  puts "Nome %s" % nome  
4  puts "Idade %d" % idade  
5  puts "Nome %s \nIdade %d" % [nome, idade]
```

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo**
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos

Controle de Fluxo (1)

if ... elsif ... else
unless
case

Controle de Fluxo (2)

- Não existe a necessidade de uso de parênteses ou chaves
- Utilize a instrução `end` no final do bloco

Listing 3: if.rb

```
1 nivel = 5
2 if nivel == 3
3   puts "basico"
4 elsif numero == 5
5   puts "avancado"
6 else
7   puts "desconhecido"
8 end
9 # => avancado
```

Listing 4: unless.rb

```
1 quantidade = 5
2 unless a == 6
3   puts "quantidade incorreta"
4 end
5 # => quantidade incorreta
```

Controle de Fluxo (3)

Listing 5: case_1.rb

```
1  idade = 21
2  case
3    when idade >= 21
4      puts "Voce pode comprar cerveja"
5    when 1 == 0
6      puts "Escrito por um programador bebado"
7    else
8      puts "Nada a dizer"
9  end
10 # Voce pode comprar cerveja
```

Controle de Fluxo (4)

Listing 6: case_2.rb

```
1 nome = 'Otello Neves'
2 case nome
3   when /neve/i then puts "Algo e suspeito aqui"
4   when 'Eduardo' then puts "Seu nome e Eduardo"
5 end
6 # Algo e suspeito aqui
```

Operadores Lógicos (em ordem de precedência)

<code><=, <, >, >=</code>	Comparação
<code>==, !=</code>	Igual ou diferente
<code>&&</code>	Conectivo e
<code> </code>	Conectivo ou

True e False

- **false** e **nil** são booleanos **FALSOS**
- Todo o restante é **VERDADEIRO**

Listing 7: true_false.rb

```
1 puts "0 e true" if 0
2 puts "false e true?" if "false"
3 puts "nao - false e false" if false
4 puts "string vazia is true" if ""
5 puts "nil e true?" if "nil"
6 puts "nao - nil is false" if nil
```

Recapitulando

- Existe muitas opções de fluxo de controle
- A formato em um linha é muito expressiva
- Exceto `nil` e `false`, os demais valores são verdadeiros.

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações**
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos

Loops e Interações (1)

loop
while e until
for
each e times

Loops e Interações (2)

■ loop

Listing 8: loop.rb

```
1 i = 0
2 loop do
3   i += 2
4   puts i
5   if i == 10
6     break
7   end
8 end
9 # 2
10 # 4
11 # 6
12 # 8
13 # 10
```

Loops e Interações (3)

■ while e until

Listing 9: while.rb

```
1 a = 10
2 while a > 9
3   puts a
4   a -= 1
5 end
6 # => 10
```

Listing 10: until.rb

```
1 a = 9
2 until a >= 10
3   puts a
4   a += 1
5 end
6 # => 9
```

Loops e Interações (4)

- for (**difícilmente empregado**)
- each/times é preferível

Listing 11: for_loop.rb

```
1 for i in 0..2
2   puts i
3 end
4 # => 0
5 # => 1
6 # => 2
```

Loops e Interações (5)

■ each

Listing 12: each_1.rb

```
1 nomes = ['Joao', 'Maria', 'Ana']
2 nomes.each { |nome| puts nome }
3 # Joao
4 # Maria
5 # Ana
```

Listing 13: each_2.rb

```
1 nomes = ['Joao', 'Maria', 'Ana']
2 n = 1
3 nomes.each do |nome|
4   puts "#{n}.#{nome}"
5   n += 1
6 end
7 # 1.Joao
8 # 2.Maria
9 # 3.Ana
```

Exercícios (1)

1. Escreva um *script* Ruby que sorteia um número de 1 a 10 e permite que o usuário tente 3 vezes até acertá-lo. A cada tentativa errada, o programa informa se o número a adivinhar está abaixo ou acima. **Dica:** utilize $\text{rand}(n) + 1$

Exercícios (2)

Listing 14: loop.rb

```
1 puts "Bem-vindo ao jogo da adivinhacao"
2 puts "Qual e o seu nome?"
3 nome = gets
4 puts "\n\n\n\n"
5 puts "Comecaremos o jogo para voce, " + nome
6 puts "Escolhendo um numero secreto entre 1 e 10..."
7 numero_secreto = rand(10) + 1
8 puts "Escolhido... "
9 puts "Que tal adivinhar hoje o nosso numero secreto?"
10
11
12 (1..3).each do | tentativa |
13   puts "\n\n\n\n"
14   puts "Tentativa %d" %tentativa
15   puts "Entre com o numero"
16   chute = gets.chomp.to_i
17   puts "áSer que acertou? êVoc chutou %d" % chute
```

Exercícios (3)

```
18
19     acertou = numero_secreto == chute.to_i
20
21     if acertou
22         puts "Acertou!"
23         break
24     else
25         maior = numero_secreto > chute.to_i
26         if maior
27             puts "O numero secreto e maior!"
28         else
29             puts "O numero secreto e menor!"
30         end
31     end
32 end
```

Recapitulando

- Existe muitas opções de loops e interações
- `each` é **preferível** ao loop `for` para percorrer arrays

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos**
- 6 Blocos
- 7 Strings
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos

Funções e Métodos

- Tecnicamente, uma **função** é definida **fora** de uma classe
- Um **método** é definido dentro de uma classe
- Em Ruby, **toda** função/método é pertence a pelo menos uma classe
 - nem sempre explicitamente escrito em uma classe

Conclusão: Toda **função** é na verdade um **método** em Ruby

Métodos

- Parênteses são **opcionais**
 - tanto para definição quanto para a chamada do método
- Usado para tornar o código mais claro

parens.rb

```
1 def soma
2   puts "sem parenteses"
3 end
4 def subtrai()
5   puts "com parenteses"
6 end
7 soma()
8 soma
9 subtrai
```

Parâmetros e Retorno

- Não é necessário declarar o tipo dos parâmetros
- O método pode retornar qualquer valor
- O comando `return` é opcional
 - o valor da **última linha** executada é retornada

return_optional.rb

```
1 def soma(um, dois)
2   um + dois
3 end
4 def divide(um, dois)
5   return "Acho que nao..." if dois == 0
6   um / dois
7 end
8 puts soma(2, 2) # => 4
9 puts divide(2, 0) # => Acho que nao...
10 puts divide(12, 4) # => 3
```

Nomes de Métodos Expressivos

- Nomes de métodos podem terminar com:
 - '?' - métodos com retorno booleano
 - '!' - métodos com efeitos colaterais

Listing 15: expressive.rb

```
1 def pode_dividir_por?(n)
2   return false if n.zero?
3   true
4 end
5 puts pode_dividir_por? 3 # => true
6 puts pode_dividir_por? 0 # => false
```

Argumentos Padrões(Defaults)

- Métodos podem ter argumentos padrões
 - se o valor é passado, ele é utilizado
 - senão, o valor padrão é utilizado

Listing 16: default_args.rb

```
1 def fatorial(n = 5)
2   n == 0? 1 : n * fatorial(n - 1)
3 end
4 puts fatorial 5    # => 120
5 puts fatorial     # => 120
6 puts fatorial(3)  # => 6
```

Quantidade Variável de Argumentos

- * prefixa o parâmetro com quantidade variável de argumentos
- Pode ser utilizado com parâmetros no início, meio e final

Listing 17: splat.rb

```
1 def max(um_parametro, *numeros, outro)
2   # os argumentos do parametro numero se
3   # tornam um array
4   numeros.max
5 end
6 puts max("algo", 7, 32, -4, "algo mais") # => 32
```

Exercícios

1. Refatore o jogo que permite o usuário adivinhar para utilizar métodos.

Recapitulando

- Não há necessidade de declarar o tipo de parâmetro passado ou retornado (linguagem dinâmica)
- return é opcional - a última linha executável é "retornada"
- Permite métodos com quantidade variável de argumentos ou argumentos padrão

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos**
- 7 Strings
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos

Blocos (1)

- Um "Trecho" de código
 - escrito entre chaves({}) ou entre **do** e **end**
 - passado para métodos como o **último** parâmetro
- **Convenção**
 - use chaves({}) quando o bloco contém uma linha
 - use **do** e **end** quando o bloco contém múltiplas linhas
- Frequentemente utilizado em **iteração**

Blocos (2)

Listing 18: times.rb

```
1 1.times { puts "Hello World!" }
2 # => Hello World!
3 2.times do |index|
4   if index > 0
5     puts index
6   end
7 end
8 # => 1
9 2.times { |index| puts index if index > 0 }
10 # => 1
```

Utilizando Blocos

- Duas técnicas para utilizar blocos nos métodos
- **Implicitamente:**
 - use `block_given?` para checar se o bloco foi passado
 - use `yield` para **chamar** o bloco
- **Explicitamente:**
 - use `&` como prefixo do último parâmetro
 - use `call` para **chamar** o bloco

Técnica Implícita

- Necessário checar com `block_given?`
 - se não uma exceção será lançada

Listing 19: implicit_blocks.rb

```
1 def imprime_duas_vezes
2   return "Nenhum bloco foi passado" unless block_given?
3   yield
4   yield
5 end
6 puts imprime_duas_vezes { print "Hello " } # => Hello
7                                           # => Hello
8 puts imprime_duas_vezes # => Nenhum bloco foi passado
```

Técnica Explícita

- Necessário checar com nil?

Listing 20: implicit_blocks.rb

```
1 def imprime_duas_vezes (&um_bloco)
2   return "Nenhum bloco foi passado" if um_bloco.nil?
3   um_bloco.call
4   um_bloco.call
5 end
6
7 puts imprime_duas_vezes # => Nenhum bloco foi passado
8 imprime_duas_vezes { puts "Hello" } # => Hello
9                                     # => Hello
```

Recapitulando

- Blocos são apenas **trechos** de códigos que podem ser passados para métodos
- Tanto explicitamente quanto implicitamente

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings**
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos

Strings (1)

- Strings com aspas simples

- permitem a utilização de ' com \
- mostra a string como foi escrita

- Strings com aspas duplas

- interpreta caracteres especiais como \n e \t
- permite a interpolação de strings, evitando concatenação

Strings (2)

Listing 21: strings.rb

```
1  aspas_simples = 'D\' Silva Filho\n programa em Ruby!'
2  aspas_duplas  = "D\' Silva Filho\n programa em Ruby!"
3  puts aspas_simples # => D' Silva Filho\n programa em Ruby!
4  puts aspas_duplas  # => D' Silva Filho\n
5                        # =>  programa em Ruby!
6  def multiplica (um, dois)
7    "#{um} multiplicado por #{dois} = #{um * dois}"
8  end
9  puts multiplica(5, 3)
10 # => 5 multiplicado por 3 = 15
```

Strings (3)

- Métodos terminados com ! modificam a string
 - a maioria retorna apenas um novo string
- Permite o uso do %Q{textos longos com multiplas linhas}
 - o mesmo comportamento de strings com aspas duplas
- É essencial dominar a API de Strings do Ruby

Strings (4)

Listing 22: more_strings.rb

```
1 nome = " tim"
2 puts nome.lstrip.capitalize # => Tim
3 p nome # => " tim"
4 nome.lstrip! # remove os espacos do inicial (modifica)
5 nome[0] = 'K' # substitui o primeiro caracter
6 puts nome # => Kim
7
8 clima = %Q{0 dia esta quente la fora
9         pegue os guarda\~chuva}
10
11 clima.lines do |line|
12   line.sub! 'quente', 'chuvoso' # substitui 'quente' with 'chuvoso'
13   puts "#{line.strip}"
14 end
15 # => dia esta quente la fora
16 # => pegue os guarda\~chuvas
```

Símbolos

- **:símbolo** — string altamente otimizadas
 - ex. :domingo, :dolar, :calcio, :id
- Constantes que não precisam ser pré-declaradas
- Garantia de **unicidade** e **imutabilidade**
- Podem ser convertidos para uma **String** com **to_s**
 - ou de **String** para **Símbolo** com **to_sym**

Exercícios (1)

1. Escreva os *script* em Ruby e verifique os resultados após a sua execução.

Listing 23: more_strings.rb

```
1 nome = " tim"
2 puts nome.lstrip.capitalize # => Tim
3 p nome # => " tim"
4 nome.lstrip! # remove os espacos do inicial (modifica)
5 nome[0] = 'K' # substitui o primeiro caracter
6 puts nome # => Kim
7
8 clima = %Q{0 dia esta quente la fora
9         pegue os guarda\-chuva}
10
11 clima.lines do |line|
```

Exercícios (2)

```
12  line.sub! 'quente', 'chuvoso' # substitui 'quente' with 'chuvoso'
13  puts "#{line.strip}"
14  end
15  # => dia esta quente la fora
16  # => pegue os guarda\ -chuvas
```


Recapitulando

- A interpolação evita a concatenação de strings
- Strings oferecem uma API muito útil

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings
- 8 Arrays**
- 9 Hashes
- 10 Classes e Objetos

Arrays (1)

- Coleção de objetos (auto-expandível)
- Indexado pelo operador (método) `[]`
- Pode ser indexado por números negativos ou intervalos
- Tipos heterogêneos são permitidos em um mesmo array
- `%{str1 str2}` pode ser utilizado para criar um array de strings

Arrays (2)

Listing 24: arrays.rb

```
1 heterogeneo = [1, "dois", :tres]
2 puts heterogeneo[1] # => dois (indice comeca em 0)
3 palavras = %w{ olhe que grande dia hoje! }
4 puts palavras[-2] # => dia
5 puts "#{palavras.first} - #{palavras.last}" # => olha - hoje!
6 p palavras[-3, 2] # => ["grande", "dia"] (volta 3 and pega 2)
7 p palavras[2..4] # => ["grande", "dia", "hoje!"]
8 puts palavras.join(',') # => olhe,que,grande,dia,hoje!
```

Arrays (3)

- Modificando arrays:
 - criação: `= []`
 - inclusão: `push` ou `jj`
 - remoção: `pop` ou `shift`
- Extração randômica de elementos com `sample`
- Classificação ou inversão com `sort!` ou `reverse!`

Arrays (4)

Listing 25: arrays2

```
1 pilha = []; pilha << "um"; pilha.push ("dois")
2 puts pilha.pop # => dois
3
4 fila = []; fila.push "um"; fila.push "dois"
5 puts fila.shift # => um
6
7 a = [5,3,4,2].sort!.reverse!
8 p a # => [5,4,3,2]
9 p a.sample(2) # => extrai dois elementos
10
11 a[6] = 33
12 p a # => [5, 4, 3, 2, nil, nil, 33]
```

Arrays (5)

■ Métodos úteis

- **each** - percorre um array
- **select** - filtra por seleção
- **reject** - filtra por rejeição
- **map** - modifica cada elemento do array

Arrays (6)

Listing 26: arrays2

```
1 a = [1, 3, 4, 7, 8, 10]
2 a.each { |num| print num } # => 1347810
3 puts # => (nova linha)
4 novo = a.select { |num| num > 4 }
5 p novo # => [7, 8, 10]
6 novo = a.select { |num| num < 10 }
7           .reject{ |num| num.even? }
8 p novo # => [1, 3, 7]
9 # Multiplica cada elemento do array produzindo
10 # um novo array
11 novo = a.map {|x| x * 3}
12 p novo # => [3, 9, 12, 21, 24, 30]
```


Exercícios (1)

1. Escreva um *script* em Ruby que receba pelo teclado um array de tamanho 5 e um número e retorne a mensagem "o número está no array" ou "o número não está no array". A verificação deverá ser feita por um método.

Recapitulando

- A API de arrays é flexível e poderosa
- Existem diversas formas de processar um elemento do array

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings
- 8 Arrays
- 9 Hashes**
- 10 Classes e Objetos

Hashes (1)

- **Coleção indexada** de objetos
- Criados com `{ }` ou **Hash.new**
- Também conhecidos como **arrays associativos**
- Pode ser indexado com **qualquer** tipo de dados
 - não apenas com **inteiros**
- Acessados utilizando o operador **[]**
- Atribuição de valores poder feita usando:
 - **=>** (criação)
 - **[]** (pós-criação)

Hashes (2)

Listing 27: hashes.rb

```
1  propiedades = { "font" => "Arial", "size" => 12, "color" => "red"}
2
3  puts propiedades.length # => 3
4  puts propiedades["font"] # => Arial
5  propiedades["background"] = "Blue"
6  propiedades.each_pair do |key, value|
7    puts "Key: #{key} value: #{value}"
8  end
9  # => Key: font value: Arial
10 # => Key: size value: 12
11 # => Key: color value: red
12 # => Key: background value: Blue
```

Hashes (3)

- E se tentarmos **acessar** um valor em Hash que **não existe**?
 - **nil** é retornado
- Se o Hash é criado com **Hash.new(0)** 0 é retornado.

Listing 28: word_frequency.rb

```
1 frequencias = Hash.new(0)
2 sentenca = "Chicka chicka boom boom"
3 sentenca.split.each do |word|
4   frequencias[word.downcase] += 1
5 end
6 puts frequencias # => {"chicka" => 2, "boom" => 2}
```

Hashes (4)

- A partir da versão 1.9
 - A ordem de criação do Hash é **mantida**
 - A sintaxe **simbolo:** pode ser utilizada, se símbolos são utilizados como chave
 - Se o Hash é o **último argumento**, {} são opcionais

Hashes (5)

Listing 29: more_hashes.rb

```
1 familia = {oldest: "Jim", older: "Joe", younger: "Jack"}
2 familia[:youngest] = "Jeremy"
3 p familia
4 # => {:oldest=>"Jim",:older=>"Joe",:younger=>"\Jack
5 # => ,:youngest => "\Jeremy}
6
7 def ajusta_cores (props = {foreground: "red",background: "white"})
8   puts "Foreground: #{props[:foreground]}" if props[:foreground]
9   puts "Background: #{props[:background]}" if props[:background]
10 end
11 ajusta_cores # => foreground: red
12             # => background: white
13 ajusta_cores ({ :foreground => "green" }) # => foreground: green
14 ajusta_cores background: "yella" # => background: yella
15 ajusta_cores :background => "magenta" # => background: magenta
```


Exercícios

1. Escreva os *script* em Ruby e verifique os resultados após a sua execução.

```
1 opostos = {positivo: "negativo"  
2 , aberto: "fechado"  
3 , direita: "esquerda"}  
4 opostos.each_key { |key| puts key }  
5 opostos.each_value { |value| puts value }  
6 opostos.each { |key, value| puts "O oposto de #{key} eh #{value}" }
```

Recapitulando

- Hashes são coleções indexadas
- Usado de forma similar aos arrays

Ruby

- 1 Introdução
- 2 Entrada e Saída
- 3 Controle de Fluxo
- 4 Loops e Interações
- 5 Funções e Métodos
- 6 Blocos
- 7 Strings
- 8 Arrays
- 9 Hashes
- 10 Classes e Objetos**

OO

- OO possibilita identificar "coisas" que serão tratadas pelo programa
- **Classes** são descrições dessas "coisas" e container de métodos
- Objetos são **instâncias** dessas classes
- Objetos contêm **variáveis de instância** (estado)

Variáveis de Instância

- Iniciam com: @
 - exemplo: @nome
- Não há necessidade de declará-las
- Disponível para todas as instâncias dos métodos da classe

Criação de Objetos (1)

- Classes são fábricas
 - **new** cria uma instância da classe e invoca o método **initialize**
 - O estado do objeto deve ser inicializado no método **initialize** (construtor)

Criação de Objetos (2)

Listing 30: classes.rb

```
1 class Pessoa
2   def initialize (nome, idade)
3     @nome = nome
4     @idade = idade
5   end
6   def get_info
7     "Nome: #{@nome}, age: #{@idade}"
8   end
9 end
10
11 pessoa1 = Pessoa.new("Jose", 14)
12 p pessoa1.instance_variables # [:@nome, :@idade]
13 puts pessoa1.get_info # => Nome: Jose, idade: 14
```

Acesso a Variáveis de Instância (1)

- Variáveis de instância são **privadas**
- Métodos são públicos por padrão
- Getters/setters para acessar variáveis de instância são necessários

Acesso a Variáveis de Instância (2)

Listing 31: instance_vars.rb

```
1 class Pessoa
2   def initialize (nome, idade)
3     @nome = nome
4     @idade = idade
5   end
6   def nome
7     @nome
8   end
9   def nome= (novo_nome)
10    @nome = novo_nome
11  end
12 end
13 pessoa1 = Pessoa.new("Jose", 14)
14 puts pessoa1.nome # Jose
15 pessoa1.nome = "Maria"
16 puts pessoa1.nome # Maria
17 # puts pessoa1.idade # undefined method 'idade' for #<Pessoa:
```

Acesso a Variáveis de Instância (3)

- Muitas vezes as lógicas dos getters/setters são muito simples
- Existe uma maneira mais fácil de definir esses métodos em Ruby
 - `attr_accessor` - getter e setter
 - `attr_reader` - somente getter
 - `attr_writer` - somente setter

Acesso a Variáveis de Instância (4)

Listing 32: attr_accessor.rb

```
1 class Pessoa
2   attr_accessor :nome, :idade # getters and setters for nome and id
3 end
4
5 pessoa1 = Pessoa.new
6 p pessoa1.nome # => nil
7 pessoa1.nome = "Maria"
8 pessoa1.idade = 15
9 puts pessoa1.nome # => Mike
10 puts pessoa1.idade # => 15
11 pessoa1.idade = "quinze"
12 puts pessoa1.idade # => fifteen
```

- **Dois problemas** com o exemplo acima:
 - Pessoa se encontra em um estado não inicializado na criação

Acesso a Variáveis de Instância (5)

- Algumas vezes é necessário controlar, por exemplo, a idade atribuída
- **Solução:** use o construtor de forma mais inteligente utilizando o comando **self**

Listing 33: self.rb

```
1 class Pessoa
2   attr_reader :idade
3   attr_accessor :nome
4
5   def initialize (nome, idade) # CONSTRUCTOR
6     @nome = nome
7     self.idade = idade # call the idade= method
8     puts idade
9   end
```

Acesso a Variáveis de Instância (6)

```
10  def idade= (nova_idade)
11      @idade ||= 5
12      @idade = nova_idade unless nova_idade > 120
13  end
14  end
15
16  pessoa1 = Pessoa.new("Kim", 13) # => 13
17  puts "Minha idade e #{pessoa1.idade}" # => Minha idade e 13
18  pessoa1.idade = 130 # Tenta mudar a idade
19  puts pessoa1.idade # => 13 o setter ão permite
```

Métodos e Variáveis de Classe (1)

- **Existem** três maneiras para definir métodos de classe
- Variáveis de classe começam com **@@**

Listing 34: class_methods_and_variables.rb

```
1 class MathFunctions
2   def self.double(var) # 1. Usando self
3     times_called; var * 2;
4   end
5   class << self # 2. Usando << self
6     def times_called
7       @@times_called ||= 0; @@times_called += 1
8     end
9   end
10 end
11 def MathFunctions.triple(var) # 3. Fora da classe
```

Métodos e Variáveis de Classe (2)

```
12     times_called; var * 3
13 end
14 puts MathFunctions.double 5 # => 10
15 puts MathFunctions.triple(3) # => 9
16 puts MathFunctions.times_called # => 3
```

Herança de Classes (1)

Listing 35: inheritance.rb

```
1 class Cao # implicitamente herda de Object
2   def to_s
3     "Cao"
4   end
5   def late
6     "late alto"
7   end
8 end
9 class CaoPequeno < Cao
10   def late # Override
11     "late baixo"
12   end
13 end
14 cao = Cao.new
15 cao_pequeno = CaoPequeno.new
```


Herança de Classes (2)

```
16 puts "#{cao}1 #{cao.late}"  
17 puts "#{cao_pequeno}2 #{cao_pequeno.late}"
```

Exercícios (1)

Elabore na linguagem Java os códigos para os seguintes requisitos:

1. Escreva a classe **Sobremesa** com *getters* e *setters* para os atributos nome e calorias. O construtor dessa classe deverá receber como parâmetros nome e calorias.
2. Defina as operações de instância *eh Saudavel*, que retorna true se e somente se a sobremesa tem menos de 200 calorias, e *eh Deliciosa*, que retorna true para todas as sobremesas.

Exercícios (2)

3. Crie a classe **GeleiaEmCompota** que herdará da classe **Sobremesa**. O seu construtor deverá aceitar um único argumento denominado sabor; a sua quantidade padrão de calorias é 5 e seu nome deverá ser precedido de "Geléia em Compota de ", por exemplo, "Geléia em Compota de Morango".
4. Inclua um *getter* and *setter* para o atributo sabor.
5. Modifique a operação *ehDeliciosa* para retornar **false** se o sabor é alcaçuz e **true** para todos os outros sabores. O comportamento dessa operação para sobremesas que não são geléias em compotas não devem ser alterados.

Recapitulando

- Objetos são criados com `new`
- Utilize o `attr_` para criar getters/setters
- Não se esqueça do `self` quando necessário
- Variáveis de classe são definidas com `@@`