# **Ruby Avançado I**

#### Assunto da aula

A partir desta aula o conteúdo do curso segue para conceitos avançados da linguagem Ruby e para dar início a este ciclo você conhecerá algumas das coisas mais legais:  blocks, lambdas, namespaces e modules.

#### Conteúdo

* Blocks
* Lambdas
* Namespaces
* Modules

#### Blocks

Um bloco pode ser entendido como uma função anônima, ou seja, uma função sem nome. É definido entre **do**..**end**  ou **colchetes** e da mesma forma que os métodos que você viu até agora, pode receber parâmetros para sua execução.

Use **colchetes**para escrever blocos que ocupam apenas uma linha.

1- Em um arquivo chamado **block.rb** adicione o seguinte código:

|  |  |
| --- | --- |
| 1 | 5.times { puts "Exec the block" } |

2- Utilize barras verticais para passar parâmetros para um bloco

|  |  |
| --- | --- |
| 1  2  3  4 | sum = 0  numbers = [5, 10, 5]  numbers.each {|number| sum += number }  puts sum |

3- Em blocos que ocupam várias linhas, faça uso do **do..end**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | foo = {2 => 3, 4 => 5}    foo.each do |key, value|   puts "key = #{key}"   puts "value = #{value}"   puts "key \* value = #{key \* value}"   puts '---'  end |

4- Um bloco pode ser passado como  argumento implícito de um método.

Depois, para chamar dentro do método o bloco que foi passado utilize a palavra **yield**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def foo   # Call the block   yield   yield  end    foo { puts "Exec the block"} |

Perceba que ao chamar o método, o bloco será executado duas vezes.

5- E se o bloco for opcional?

O ruby oferece um método chamado **block\_given?**para verificar se o bloco foi passado como argumento

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | def foo   if block\_given?     # Call the block     yield   else     puts "Sem parâmetro do tipo bloco"   end  end    foo  foo { puts "Com parâmetro do tipo bloco"} |

Note que você criou um método que executa um trecho de código se o bloco for passado como argumento e outro trecho caso não seja.

6- Outra forma de receber blocos como parâmetro é utilizar o símbolo **&**

Ex:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def foo(name, &block)   @name = name   block.call  end    foo('Leonardo') { puts "Hellow #{@name}" } |

Para executar um bloco recebido desta forma é necessário apenas utilizar o nome do bloco acompanhado pelo método **call.**

Outra dica importante é sempre deixar o **&nome\_do\_bloco** como último parâmetro a ser recebido pelo método

7- Você também pode passar um bloco que ocupa várias linhas como parâmetro

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def foo(numbers, &block)   if block\_given?     numbers.each do |key, value|       block.call(key, value)     end   end  end    numbers = { 2 => 2, 3 => 3, 4 => 4 }      foo(numbers) do |key, value|   puts "#{key} \* #{value} = #{key \* value}"   puts "#{key} + #{value} = #{key + value}"   puts "---"  end |

Ao chamar o bloco  você passou os parâmetros que ele precisa para ser executado

#### Lambda

Bastante similar aos blocos com as seguintes diferenças:

**Sintaxe**

Uma lambda é iniciada com a palavra **lambda.**

1- Crie um arquivo chamado **lambda.rb**com o seguinte código

|  |  |
| --- | --- |
| 1  2 | first\_lambda = lambda { puts "my first lambda"}  first\_lambda.call |

Ela pode ser guardada em uma variável para ser chamada futuramente com o método **call**.

2- Você pode abreviar a declaração de uma lambda da seguinte forma

|  |  |
| --- | --- |
| 1  2 | first\_lambda = -> { puts "my first lambda"}  first\_lambda.call |

3- Uma lambda também pode receber parâmetros para sua execução

|  |  |
| --- | --- |
| 1  2  3  4  5 | first\_lambda = -> (names){ names.each { |name |puts name} }    names = ["joão", "maria", "pedro"]    first\_lambda.call(names) |

Perceba que você executou um block dentro de uma lambda

4- Em lambdas que ocupam várias linhas, não declare a lambda de forma abreviada e utilize o **do..end**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | my\_lambda = lambda do |numbers|   index = 0   puts 'Número atual + Próximo número'   numbers.each do |number|     return if numbers[index] == numbers.last     puts "(#{numbers[index]}) + (#{numbers[index + 1]})"     puts numbers[index] + numbers[index + 1]     index += 1   end  end    numbers = [1, 2, 3, 4]    my\_lambda.call(numbers) |

**Argumentos**

1- Diferente de blocks, você pode passar mais de uma lambda como argumentos de um método.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def foo(first\_lambda, second\_lambda)   first\_lambda.call   second\_lambda.call  end    first\_lambda = lambda { puts "my first lambda"}  second\_lambda = lambda { puts "my second lambda"}    foo(first\_lambda, second\_lambda) |

**Modules – Namespaces**

1- Crie um programa chamado namespace.rb e coloque nele:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | module ReverseWorld      def self.puts text          print text.reverse.to\_s      end  end    ReverseWorld::puts 'O resultado é'  puts 'O resultado é' |

2 – Execute o programa rodando:

|  |  |
| --- | --- |
| 1 | ruby namespace.rb |

#### Modules – Mixins

1- Crie um programa chamado mixins.rb e coloque nele:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | module ImpressaoDecorada      def imprimir text          decoracao = '#' \* 100          puts decoracao          puts text          puts decoracao      end  end    module Pernas      include ImpressaoDecorada        def chute\_frontal          imprimir 'Chute Frontal'      end        def chute\_lateral        imprimir 'Chute Lateral'      end  end    module Bracos      include ImpressaoDecorada        def jab\_de\_direita          imprimir 'Jab de direita'      end        def jab\_de\_esquerda          imprimir 'Jab de esquerda'      end        def gancho        imprimir 'Gancho'      end  end    class LutadorX      include Pernas      include Bracos  end    class LutadorY      include Pernas  end    lutadorx = LutadorX.new  lutadorx.chute\_frontal  lutadorx.jab\_de\_direita    lutadory = LutadorY.new  lutadory.chute\_lateral |

2 – Execute o programa rodando:

|  |  |
| --- | --- |
| 1 | ruby namespace.rb |

Nós criamos um module ImpressaoDecorada e incluimos ele nos outros dois modules (pernas e braços), depois incluimos estes últimos dois modules nas classes LutadorX e LutadorY.

#### Missões especiais

As missões desta aula reforçarão seu aprendizado e ajudarão a expandir sua visão sobre as possibilidades do uso de modules.

**Missão 1**

O ruby oferece um método chamado **capitalize**para tornar a primeira letra de uma string maiúscula.

Sabendo disso crie uma lambda que recebe um nome como parâmetro e o imprime com a primeira letra maiúscula. Esta lambda deverá ser salva dentro de uma variável que será passada como argumento de um método chamado **capitalize\_name**.

Dentro deste método você chamará a lambda duas vezes, passando como parâmetro em cada uma delas um nome diferente.

**Missão 2**

Crie um módulo chamado **Person**com as classes **Juridic** e **Physical**.

Ao executar a instrução

|  |  |
| --- | --- |
| 1 | Person::Juridic.new('M. C. Investimentos', '4241.123/0001').add |

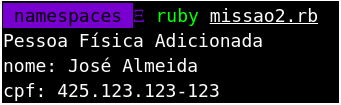
Seu código deverá retornar



E com a instrução

|  |  |
| --- | --- |
| 1 | Person::Physical.new('José Almeida', '425.123.123-123').add |

Deverá retornar



**Missão 3**

Pesquisar o que é o Proc em Ruby e descobrir as diferenças entre ele e o Lambda