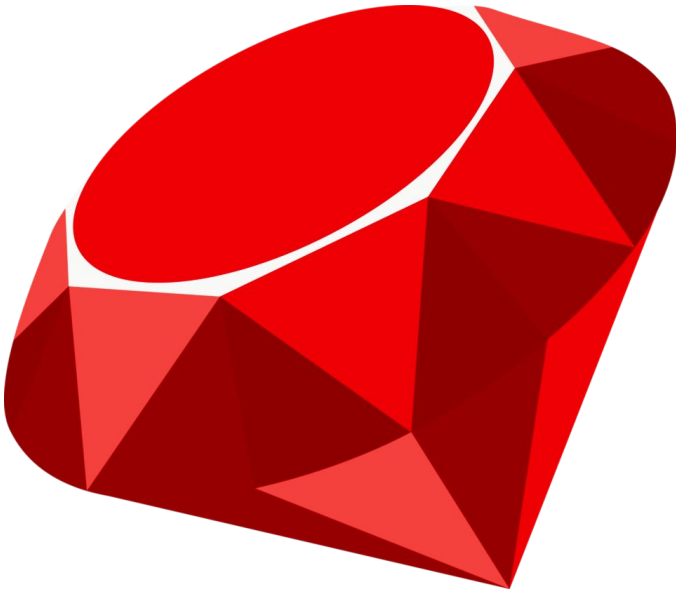
Ruby

Fernanda Oliveira
Patrícia Wang
Vanessa Soares



Índice

HISTÓRIA

- Criada no Japão por Yukihiro Matsumoto (Matz)
- Lançada no Japão em dezembro/1995
- Lançada oficialmente em dezembro/1996

HISTÓRIA

- Matz queria uma linguagem
  - Mais poderosa que Perl
  - Mais orientada a objetos que Python
  - Que tivesse tudo o que ele sempre amou em Lisp, Eiffel e Smalltalk
  - Funcional como Lisp, Haskell e Scheme.

Ruby on Rails

Curiosidade

- O nome "Ruby", foi decidido durante uma conversa online entre Matz e um amigo em 1993, antes que qualquer linha de código tivesse sido escrita para a linguagem.
- Coral x Ruby

Características

- Ruby é uma linguagem de programação
  - Interpretada
  - Open-Source
  - Orientada a Objetos
  - Funcional
  - Dinâmica
  - Altamente portável

Tudo em Ruby é objeto!

- Matz decidiu trabalhar com uma linguagem mais orientada a objeto que Python.
- Ruby é uma linguagem orientada a objeto pura! (Smalltalk)
- Os programadores definem classes de objetos em suas aplicações para imitar (ou simular) o mundo real.

Tudo em Ruby é objeto!

```
2.2.1 :001 > 10.class
 => Fixnum
2.2.1 :002 > class Person
2.2.1 :003?>   end
 => nil
2.2.1 :004 > p = Person.new
 => #<Person:0x9a2b3b4>
2.2.1 :005 > p.class
 => Person
2.2.1 :006 > Person.class
 => Class
```

Tipos Básicos

- Não temos tipos primitivos, somente abstratos.
- Todos tem comportamento de objetos.

- Tipos
  - Fixnums
  - Bignums
  - Floats
  - String
  - Boolean

Tipos Abstratos de Dados

```
2.2.1 :104 >   1.class
 => Fixnum
2.2.1 :105 > (1024**30).class
 => Bignum
2.2.1 :106 > 1.45.class
 => Float
2.2.1 :107 > "ola mundo".class
 => String
2.2.1 :108 > true.class
 => TrueClass
2.2.1 :109 > false.class
 => FalseClass
2.2.1 :110 >
```

Tipagem

- Dinâmicamente tipada

```
1   x = 100
2   (1..10).each{ x = x * 1000 ;  puts "#{x.class} #{x}"  }
3
4   #Fixnum 100000
5   #Fixnum 100000000
6   #Fixnum 100000000000
7   #Fixnum 100000000000000
8   #Fixnum 100000000000000000
9   #Bignum 100000000000000000000
10  #Bignum 100000000000000000000000
11  #Bignum 100000000000000000000000000
12  #Bignum 100000000000000000000000000000
13  #Bignum 100000000000000000000000000000000
```

Tipagem

- Implicitamente tipada

```
2.2.1 :140 >   x = 20
 => 20
2.2.1 :141 > x.class
 => Fixnum
2.2.1 :142 > y = "oi"
 => "oi"
2.2.1 :143 > y.class
 => String
2.2.1 :144 >
```

Tipagem

- Fortemente tipada

```ruby
a = 100
b = "Ruby"
a + b

# `+': String can't be coerced into Fixnum (TypeError)
```

Operadores em Ruby

Operadores do Ruby (ordem decrescente de precedência)		
Método	**Operador**	**Descrição**
S	[ ] [ ]=	Referência para elemento, conjunto de elementos
S	**	Exponenciação
S	! ~ + -	Negação, complemento, operador unário
S	* / %	Multiplicação, divisão e módulo
S	+ -	Mais e menos
S	>> <<	Deslocamento à direita e à esquerda
S	&	`and' bit a bit
S	^ |	`or' exclusivo e regular bit a bit
S	<= < > >=	Operadores de comparação
S	<=> == === != =~ !~	Operadores de igualdade e padrão de jogo (!= E !~ Não podem ser definidos como métodos)
N	&&	`and' lógico
N	||	`or' lógico
N	.. ...	Range (inclusivo e exclusivo)
N	? :	if-then-else ternário
N	= %= { /= -= += |= &= >>= <<= *= &&= ||= **=	Assignment
N	defined?	Verifica se o símbolo é definido
N	not	Negação lógica
N	or and	Composição lógica
N	if unless while until	Modificadores de expressão
N	begin/end	Bloco de expressão

If.. else

```
x = 1
if x > 2 then
    puts “x is greater than 2”
elsif  x <= 2  and x != 0 then
    puts “x is 1”
else
    puts "I can't guess the
number"
end
# x is 1
```

```
x = 1
unless x  > 2 then
  puts “ x is less than 2”
else
  puts “ x is greater than 2”
end
# x is less than 2
```

While

```
i = 0
while i < 5 do
  puts i
  i += 1
end
```

```
i = 0
num = 5
begin
  puts “inside the loop i = #{i}”
  i += 1
end while i < num
```

until

```
i = 0
until i > 5 do
  puts i
  i += 1
end
```

```
i = 0
num = 5
begin
  puts “inside the loop i = #{i}”
  i += 1
end until i > num
```

For

```
for i in (0..5) do
  puts “inside the loop i = #{i}”
end
```

```
(0..5).each do |i|
    puts “inside the loop i = #{i}”
end
```

Switch

```
age =  5
case age
when 0 .. 2
   puts "baby"
when 3 .. 6
   puts "little child"
when 7 .. 12
   puts "child"
when 13 .. 18
   puts "youth"
else
   puts "adult"
end

#little child
```

Escopo

- O escopo de uma variável é o contexto no qual ela é visível ao programa.
- Nem todas as variáveis são acessíveis a todas as partes de um programa Ruby o tempo todo.

Variáveis

- **Variável local:** Variável declarada dentro de um método, só existindo dentro dos limites daquele método específico.**Exemplo:**

```
def myMethod
    age = 12
end
```

- **Variável global:** Variável que uma vez criada, é acessível em qualquer parte do programa. Simplesmente inicia-se a variável com um $.
  Exemplo: *$auxGlobal = 10*

Variáveis

- **Variável de instância:** Usar @ antes de uma variável para mostrar que ela é uma variável de instância. Isso significa que a variável está relacionada à instância da classe.
  Exemplo: *@auxInstance*

- **Variável de Classe:** São como variáveis de instância, mas ao invés de pertencer a uma instância de uma classe, elas pertencem à própria classe. Variáveis de classe sempre são iniciadas com duas @s.
  Exemplo: *@@auxClass*

Variáveis

- **Constantes:** Constantes são como variáveis. Exceto que você avisa a Ruby que seu valor é supostamente fixo. O nome de uma constante começa como uma letra maiúscula seguida de caracteres de nome.
  Exemplo: *PI = 3.1416*

Palavras reservadas

alias	defined	for	redo	undef
and	do	if	rescue	unless
BEGIN	else	in	retry	until
begin	elsif	module	return	when
break	END	next	self	while
case	end	nil	super	yield
class	ensure	not	then	
def	false	or	true	

Matriz

```ruby
require 'matrix'
matriz_1 = Matrix[ [25, 93], [-1, 66] ]
#Matrix[[25, 93], [-1, 66]]

matriz_2 = Matrix.build(3) { |row, col| col - row }
#Matrix[[0, 1, 2], [-1, 0, 1], [-2, -1, 0]]

matriz_3 = Matrix.build(2) { |row, col| col + row }
#Matrix[[0, 1], [1, 2]]

matrix_i = Matrix.identity(2)
#Matrix[[1, 0], [0, 1]]

Matrix.vstack(matriz_1, matrix_i)
#Matrix[[25, 93], [-1, 66], [1, 0], [0, 1]]

matriz_1 * matriz_3
#Matrix[[93, 211], [66, 131]]

matriz_2.transpose
#Matrix[[0, -1, -2], [1, 0, -1], [2, 1, 0]]
```

Array

- Arrays em Ruby são instâncias da classe Array, não sendo simplesmente uma estrutura de dados, mas possuindo diversos métodos auxiliares que nos ajudam no dia-a-dia.
- Um novo array pode ser criado usando o construtor literal []. Arrays podem conter diferentes tipos de objetos.

Array

```ruby
ary = [1, "two", 3.0]
#=> [1, "two", 3.0]

ary = Array.new     #=> []
Array.new(3)        #=> [nil, nil, nil]
Array.new(3, true) #=> [true, true, true]


Array.new(4) { Hash.new } #=> [{}, {}, {}, {}]


empty_table = Array.new(3) { Array.new(3) }
#=> [[nil, nil, nil], [nil, nil, nil], [nil, nil, nil]]

Array({:a => "a", :b => "b"}) #=> [[:a, "a"], [:b, "b"]]

```

Array

```ruby
lista = Array.new
lista << "RR-71"
lista << "RR-75"
lista << "FJ-91"

puts lista.size
# => 3

# ==================================================

arr = [1, 2, 3, 4, 5, 6]
arr[2]    #=> 3
arr[100]  #=> nil
arr[-3]   #=> 4
arr[2, 3] #=> [3, 4, 5]
arr[1..4] #=> [2, 3, 4, 5]
arr[1..-3] #=> [2, 3, 4]

arr.at(0) #=> 1

# ==================================================

lista = [1, 2, "string", :simbolo, /$regex^/]
puts lista[2]
# => string

lista = ["rails", "rake", "ruby", "rvm"]
lista.each do |programa|
  puts programa
end
```

Array

```ruby
funcionarios = ["Guilherme", "Sergio", "David"]
nomes_maiusculos = []

for nome in funcionarios
  nomes_maiusculos << nome.upcase
end

funcionarios = ["Guilherme", "Sergio", "David"]
nomes_maiusculos = []

funcionarios.each do |nome|
  nomes_maiusculos << nome.upcase
end

#=====================================================================

arr = ['a', 'b', 'c', 'd', 'e', 'f']
arr.fetch(100) #=> IndexError: index 100 outside of array bounds: -6...6
arr.fetch(100, "oops") #=> "oops"
arr.first # => a
arr.last # => f

arr.take (3) # => [a, b, c ]
arr.drop (3) # => [d, e, f]

#=====================================================================

browsers = ['Chrome', 'Firefox', 'Safari', 'Opera', 'IE']
browsers.length # => 5
browsers.count # => 5

browsers.empty? # => False
browsers.include? ('Konqueror') # => false

```

Array

```ruby
arr = [1, 2, 3, 4]
arr.push (5) # => [1, 2, 3, 4, 5]
arr << # 6 => [1, 2, 3, 4, 5, 6]

num = [1, 2, 3, 4, 5, 6]
num.pop # => 6
num # => [1, 2, 3, 4, 5]

numeros = [1, 2, 3, 4, 5]
numeros.each { |a| print a -= 10, " " }
# prints: -9 -8 -7 -6 -5
#=> [1, 2, 3, 4, 5]

#=============================================================================

words = %w[first second third fourth fifth sixth]
str = ""
words.reverse_each { |word| str += "#{word} " }
p str #=> "sixth fifth fourth third second first "

#=============================================================================

arr = [1, 2, 3, 4, 5, 6]
arr.select { |a| a > 3 }      #=> [4, 5, 6]
arr.reject { |a| a < 3 }      #=> [3, 4, 5, 6]
arr.drop_while { |a| a < 4 } #=> [4, 5, 6]
arr                           #=> [1, 2, 3, 4, 5, 6]

#=============================================================================

[ 1, 2 ] << "c" << "d" << [ 3, 4 ]
#=>  [ 1, 2, "c", "d", [ 3, 4 ] ]
```

Array

```ruby
a = [1, 2, 3, 4]
a.combination(1).to_a  #=> [[1],[2],[3],[4]]
a.combination(2).to_a  #=> [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
a.combination(3).to_a  #=> [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
a.combination(4).to_a  #=> [[1,2,3,4]]
a.combination(0).to_a  #=> [[]] # one combination of length 0
a.combination(5).to_a  #=> []   # no combinations of length 5

#=============================================================================

scores = [ 97, 42, 75 ]
scores.delete_if {|score| score < 80 }   #=> [97]

#=============================================================================

fill(obj) → ary
fill(obj, start [, length]) → ary
fill(obj, range ) → ary
fill { |index| block } → ary
fill(start [, length] ) { |index| block } → ary
fill(range) { |index| block } → ary

#=============================================================================

a = [ "a", "b", "c", "d" ]
a.fill("x")              #=> ["x", "x", "x", "x"]
a.fill("z", 2, 2)        #=> ["x", "x", "z", "z"]
a.fill("y", 0..1)        #=> ["y", "y", "z", "z"]
a.fill { |i| i*i }       #=> [0, 1, 4, 9]
a.fill(-2) { |i| i*i*i } #=> [0, 1, 8, 27]

#=============================================================================

```

Array

```ruby
a = [ [ 1, "one"], [2, "two"], [3, "three"], ["ii", "two"] ]
a.rassoc("two")    #=> [2, "two"]
a.rassoc("four")   #=> nil

#=============================================================

a = [ "d", "a", "e", "c", "b" ]
a.sort                      #=> ["a", "b", "c", "d", "e"]
a.sort { |x,y| y <=> x }  #=> ["e", "d", "c", "b", "a"]

#=============================================================

a = [ "a", "a", "b", "b", "c" ]
a.uniq   # => ["a", "b", "c"]

```

Hash

- São Arrays indexados, com chaves e valores, e essas chaves podem ser de qualquer tipo.

```
2.2.1 :098 > hash = {"fixnum" => 30,  1=>"um", true => "true"}
 => {"fixnum"=>30, 1=>"um", true=>"true"}
2.2.1 :099 > hash["fixnum"]
 => 30
2.2.1 :100 > hash[1]
 => "um"
2.2.1 :101 > hash[true] = 90
 => 90
2.2.1 :102 > hash
 => {"fixnum"=>30, 1=>"um", true=>90}
2.2.1 :103 >
```

Passagem de parâmetros

- Passagem por atribuição
  - todos os valores são objetos
  - valor do parâmetro real é atribuído para o parâmetro formal.
  - funciona como se fosse a passagem por referência.

Classe e Método

- Sintaxe da Classe:

```
class NewClass
    #code
end
```

- Sintaxe do Método:

```
class NewClass
    def newMethod
        #code
    end
end
```

Características dos métodos

- Os métodos sempre retornam algo, nem que seja nil:

```
def myMethod
end
myMethod
#nil
```

- Múltiplos retornos por métodos:

```
def myMethod
    return 1,2,3
end
myMethod
#[1, 2, 3]
```

```ruby
PERIODO = 2015.1   #CONSTANTE

class Aluno
    @@contador_aluno = 0  #VARIÁVEL DE CLASSE
    $media = 5.0          #VARIÁVEL GLOBAL

    def initialize (nome, nota1, nota2)
        @nome = nome      #VARIÁVEL DE INSTÂNCIA
        @nota1 = nota1    #VARIÁVEL DE INSTÂNCIA
        @nota2 = nota2    #VARIÁVEL DE INSTÂNCIA
        @@contador_aluno += 1
    end

    def numero_de_instancias
        return @@contador_aluno
    end

    def mediaFinalAluno
        nota = (@nota1 + @nota2) / 2
=begin
        bloco condicional mais simples em ruby: if está implícito!!
        condicao1 > condicao2 ? "se sim faça isso" : "senao faça esse no lugar"
=end
        return puts nota >= $media ? "Aluno #{@nome} foi aprovado em #{PERIODO}!!" :
                                     "Aluno #{@nome} foi reprovado em #{PERIODO}"
    end
end

fulano = Aluno.new("aluno1", 6.0, 10)
fulaninho = Aluno.new("aluno2", 3.0, 2.5)

puts "Número de instâncias: #{fulaninho.numero_de_instancias}"
#Número de instâncias: 2

fulano.mediaFinalAluno
#Aluno fulano foi aprovado em 2015.1!!

fulaninho.mediaFinalAluno
#Aluno fulaninho foi reprovado em 2015.1
```

Módulo

- Caixa de ferramentas que contém um conjunto de métodos e constantes.
- Há dois tipos de métodos em módulo:
  - Método de módulo
  - Método de instância

- Sintaxe do módulo:

```
module moduleName
    #code
end
```

Módulo

- Método de módulo:

```ruby
module MeuModulo
    def self.meuMetodo
        puts "sou um método de módulo"
    end
end

MeuModulo.meuMetodo
#sou um método de módulo
```

Módulo

- Método de instância:

```ruby
module MeuModulo
    def metodo_de_instancia
        puts "Método de instância"
    end
end

class MinhaClasse
    include MeuModulo #acoplando o MeuModulo
end

minha_classe = MinhaClasse.new

puts  minha_classe.metodo_de_instancia
#Método de instância
```

Herança

- Ruby permite herança simples, isto é, uma classe pode herdar os atributos e métodos de apenas uma única classe.
- Permite sobrescrever métodos da classe pai ou apanhar o método da classe pai através do *super.*

- Sintaxe de Herança:

```
class DerivedClass < BaseClass
    #code
end
```

Herança

- Ruby não permite herança múltipla!!
- Contudo, existem casos nos quais você quer incorporar dados ou comportamentos de várias classes em uma única classe, e Ruby permite isso através do uso de *mixins.*

Herança - Mixins

- Mixin permite utilizar classes e métodos de outros lugares através de módulos.

- O que diferencia mixin é que os módulos não tem instâncias. Porém quando declaramos variáveis de instância em um módulo e incluímos ele na classe, ele conseguirá acessar esses atributos e métodos como se fosse de uma classe herdada.

attr’s : reader, writer e accessor

- Os attr’s são responsáveis por ler/atribuir novos valores às variáveis sem necessidade de uma nova instanciação:

- **attr_reader :** Faz leitura da variável
- **attr_writer :** Atribui novo valor para a variável
- **attr_accessor:** Permite leitura e atribuição de um novo valor para a variável, substituindo o attr_reader e o attr_writer.

```ruby
module Music   #MÓDULO

    def play
        puts "i'm playing #{self.genre}"
    end
end

class Musician   #CLASSE PAI

    def getMusician
        puts "Hey, i'm a musician!"
    end

end


class DJ < Musician    #CLASSE FILHA
    include Music

    #attr_reader :genre
    #atrr_writer :genre
    attr_accessor :genre

    def initialize(genre)
        @genre = genre
    end
end

dj = DJ.new("Rock")
dj.play
#i'm playing Rock

dj.genre = "Eletrônica"  #attr_accessor :leitura e alteração de uma variável
dj.play
#i'm playing Eletrônica

dj.getMusician
#Hey, i'm a musician!
```

Sobrescrever operadores

```ruby
class MagicString < String
  def +@
    [self.upcase]
  end

  def -@
    downcase
  end

  def ~@
    [self.reverse]
    end

  def !@
    swapcase
  end
end

str = MagicString.new("This is my string!")
p +str                  #"THIS IS MY STRING!"
p ~str                  #["!gnirts ym si sihT"]
p !str                  #tHIS IS MY STRING
p (not str)             #tHIS IS MY STRING

```

Sobrescrever operadores

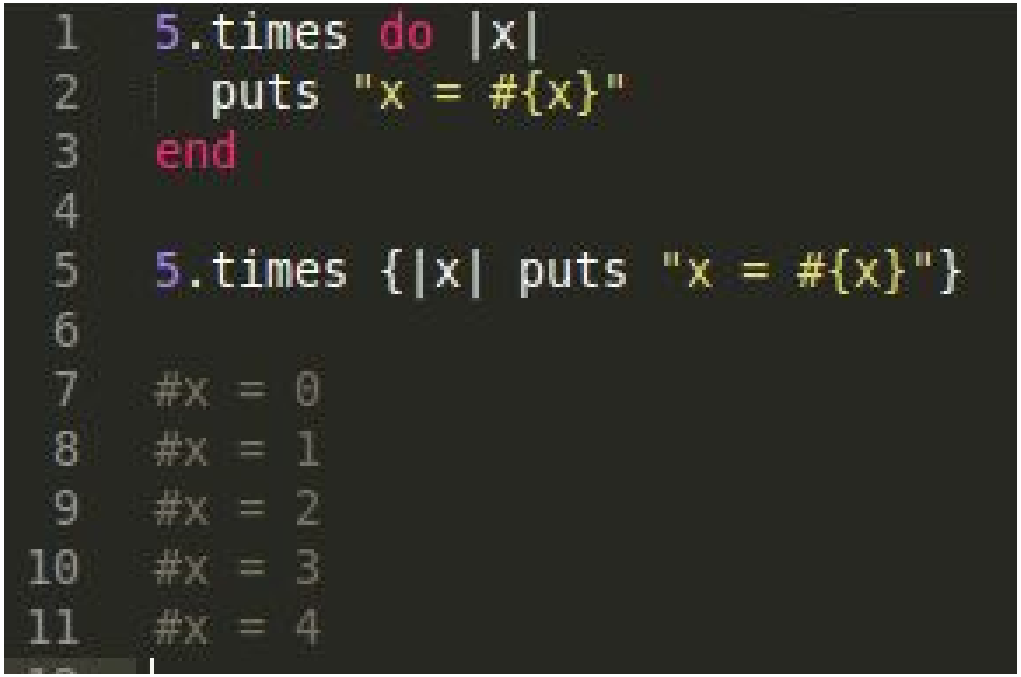
```ruby
class Produto
    attr_reader :nome, :qtd

    def initialize (nome, qtd)
        @nome = nome
        @qtd = qtd

    end

    def +(outro_produto)
        puts "sobrescrevendo operador +"
        return @qtd + outro_produto.qtd

    end


    def ==(outro_produto)
        puts "sobrescrevendo operador =="
        return @nome == outro_produto.nome
    end


    def /(outro_produto)
        puts "sobrescrevendo operador /"
        return @qtd > outro_produto.qtd
    end
end
```

Sobrescrever operadores

```
28      p Produto.new("café", 3) + Produto.new("açúcar", 2)
29      p Produto.new("café", 3) == Produto.new("açúcar", 2)
30      p Produto.new("café", 3) / Produto.new("açúcar", 2)
31
32      #sobrescrevendo operador +
33      #5
34      #sobrescrevendo operador ==
35      #false
36      #sobrescrevendo operador /
37      #true
38
```

Bloco

- Também chamados de closures.
- Pedaços de código entre chaves ou entre **do end** que você pode associar com a invocação de métodos como se fossem parâmetros.



```ruby
5.times do |x|
  puts "x = #{x}"
end

5.times {|x| puts "x = #{x}"}

#x = 0
#x = 1
#x = 2
#x = 3
#x = 4
```

Bloco

- **Yield**
  - “Ceder” o controle

```ruby
def chama_bloco
   puts 'Inicio do método'
   # você pode chamar o método com a palavra-chave yield
   yield
   yield
   puts 'Fim do metodo'
end
# Os blocos de código podem aparecer apenas no código adjacente a uma chamada de método
chama_bloco {puts 'Dentro do bloco'}

#Inicio do método
#Dentro do bloco
#Dentro do bloco
#Fim do método
```

Bloco

- **Procs != Lambdas**
  - São objetos
  - Lambdas verificam o número de argumentos
  - Cada um entende o ‘return’ de uma maneira diferente

Bloco

- **Procs != Lambdas**

```ruby
def proc_return
  p = Proc.new { return "Proc.new"}
  p.call
  return "Fim do método proc_return"
end

def lambda_return
  l = lambda { return "lambda" }
  l.call
  return "Fim do método lambda_return"
end

puts proc_return          #Proc.new
puts lambda_return        #Fim do método lambda_return
```

Exceções

- Uma exception é um tipo especial de objeto que estende ou é uma instância da classe Exception.
- Lançar uma exception significa que algo não esperado ou errado ocorreu no fluxo do programa.
- Raising é a palavra usada em ruby para lançamento de exceptions.
- Para tratar uma exception é necessário criar um código a ser executado caso o programa receba o erro. Para isso existe a palavra rescue.

Exceções

- Exceptions comuns
- A lista abaixo mostra as exceptions mais comuns em ruby e quando são lançadas, todas são filhas de Exception.

  - RuntimeError : É a exception padrão lançada pelo método raise.
  - NoMethodError : Quando um objeto recebe como parâmetro de uma mensagem um nome de método que não pode ser encontrado.
  - NameError : O interpretador não encontra uma variável ou método com o nome passado.
  - TypeError : Um método recebe como argumento algo que não pode tratar.
  - ArgumentError : Causada por número incorreto de argumentos.

Exceções

```
1  print "Digite um número:"
2  numero = gets.to_i
3
4  begin
5      resultado = 100 / numero
6      rescue
7          puts "Número digitado inválido!"
8          exit
9  end
10 puts "100/#{numero} é #{resultado} "
11
12
13 #=> Digite um número:k
14 #=> Número digitado inválido!
15
```

Exceções

```ruby
def verifica_idade(idade)
    unless idade > 18
        raise ArgumentError,
        "Você precisa ser maior de idade para jogar jogos violentos."
    end
end

verifica_idade(17)

#=> /home/ubuntu/workspace/ruby/teste.rb:3:in `verifica_idade':
#=> Você precisa ser maior de idade para jogar jogos violentos. (ArgumentError)
#=> from /home/ubuntu/workspace/ruby/teste.rb:7:in `<main>'
```

Exceções

```ruby
def levanta_e_resgata
    begin
        puts "Estou antes do raise."
        raise "Um erro ocorreu."
        puts "Estou depois do raise."
        rescue
        puts "Fui resgatado."
    end
    puts "Estou depois do bloco begin."
end

levanta_e_resgata

#=> Estou antes do raise.
#=> Fui resgatado.
#=> Estou depois do bloco begin.
```

Exceções

- Throw e catch
  - Ruby possui também throw e catch que podem ser utilizados com símbolos e a sintaxe lembra a de Erlang, onde catch é uma função que, se ocorrer algum throw com aquele label, retorna o valor do throw atrelado:

Exceções

```ruby
def pesquisa_banco(nome)
  if nome.size < 10
    throw :nome_invalido, "Nome invalido, digite novamente"
  end
  # executa a pesquisa
  "cliente #{nome}"
end

def executa_pesquisa(nome)
  catch :nome_invalido do
    cliente = pesquisa_banco(nome)
    return cliente
  end
end

puts executa_pesquisa("ana")
# => "Nome invalido, digite novamente"

puts executa_pesquisa("guilherme silveira")
# => cliente guilherme silveira

```

Exceções

- Se você quer verificar uma exceção resgatada (em que se usou o rescue). você pode mapear o objeto **Exception** para uma variável com uma cláusula rescue.

```ruby
def levanta_e_resgata
    begin
        puts "Estou antes do raise."
        raise "====== Um erro ocorreu.======="
        puts "Estou depois do raise."

        rescue Exception => e
        puts "Fui resgatado."
        puts e.message

    end
    puts "Estou depois do bloco begin."
end

levanta_e_resgata

#=> Estou antes do raise.
#=> Fui resgatado.
#=> ====== Um erro ocorreu.=======
#=> Estou depois do bloco begin.
```

Garbage Collector

- Vantagens
  - Torna a programação mais rápida e menos complicada
  - Memory Leak: desaparece!

- Desvantagens
  - menor controle sobre a gerência de memoria
  - perde-se velocidade
  - é difícil saber quando e em que ordem os objetos serão destruídos

Metaprogramação

- Por ser uma linguagem dinâmica, Ruby permite adicionar outros métodos e operações aos objetos em tempo de execução. Meta-programação é a capacidade de gerar/alterar código em tempo de execução.

Metaprogramação

➔ Imagine que tenho uma pessoa:

O que aconteceria, se eu tentasse invocar um método inexistente nesse objeto?

```
1  pessoa = Object.new()
2  pessoa.fala()
3
4  #/home/ubuntu/workspace/ruby/teste.rb:2:in `<main>':
5  #undefined method `fala' for #<Object:0x000000024d88f0> (NoMethodError)
```

Metaprogramação

- Mas e se, em tempo de execução, adicionar o comportamento (ou método) ***fala*** para essa pessoa.
- Agora que tenho uma pessoa com o método fala, posso invocá-lo:

```
1  pessoa = Object.new()
2
3  def pessoa.fala()
4    puts "Sei falar"
5  end
6  pessoa.fala()
7
8  #=>Sei falar
9
```

Metaprogramação

- Levando o dinamismo de Ruby ao extremo, podemos criar métodos que definem métodos em outros objetos:

```ruby
class Aluno
  # nao sabe nada
end

class Professor
  def ensina(aluno)
    def aluno.escreve
      puts "sei escrever!"
    end
  end
end

juca = Aluno.new
puts juca.respond_to? :escreve
# => false

professor = Professor.new
professor.ensina juca
juca.escreve
# => "sei escrever!"
```

Metaprogramação

- A criação de métodos acessores é uma tarefa muito comum no desenvolvimento orientado a objetos.

```ruby
class Pessoa
  attr_accessor :nome
end

p = Pessoa.new
p.nome = "Joaquim"
puts p.nome
# => "Joaquim"
```

- A chamada do método de classe *attr_acessor*, define os métodos *nome* e *nome =* na classe Pessoa.

Metaprogramação

- Outro exemplo interessante de metaprogramação é como definimos a visibilidade dos métodos em Ruby. Por padrão, todos os métodos definidos em uma classe são públicos, ou seja, podem ser chamados por qualquer um. Não existe nenhuma palavra reservada (keyword) da linguagem para mudar a visibilidade.

- Isto é feito com um método de classe. Toda classe possui os métodos **private**, **public** e **protected**, que são métodos que alteram outros métodos, mudando a sua visibilidade (**código alterando código == metaprogramação**).

Metaprogramação

- O método de classe private altera a visibilidade de todos os métodos definidos após ter sido chamado. Isso pode lembrar um pouco C++, que define regiões de visibilidade dentro de uma classe (seção pública, privada, ...)
- Caso seja necessário, o método public faz com que os métodos em seguida voltem a ser públicos:

```
1   class Pessoa
2
3       private
4       def metodo_privado
5           # ...
6       end
7
8       public
9       def sou_um_metodo_publico
10          #...
11      end
12  end
```

Syntax Sugar

- melhorar a legibilidade.

```ruby
class Pessoa

        attr_accessor :nome
    def initialize(nome, idade)
     @nome = nome
     @idade = idade
    end

end

    nd = Pessoa.new('Joao', 20)
    #nd.name=('Matheus')
    nd.nome = 'Matheus'
    puts nd.nome
```

Conclusão

- Ruby é uma linguagem simples e de fácil aprendizado, apesar de ter muitas funcionalidades semelhantes. No entanto, o objetivo de Matsumoto não era dificultar a vida do usuário e sim deixá-lo livre para escolher a melhor forma de implementar a resolução do problema. Assim, com a ajuda de Ruby on Rails, a linguagem acabou se popularizando já que Rails mostra o que tem de melhor no Ruby, o que acabou dando um foco na linguagem.

Referências

- http://ruby-doc.org/
- http://guru-sp.github.io/tutorial_ruby
- http://www.showthecode.com.br/2014/06/modulos-e-mixins.html
- http://nomedojogo.com/2009/12/24/classes-sao-modulos-no-ruby/
- https://www.caelum.com.br/apostila-ruby-on-rails/a-linguagem-ruby/

KEEP CALM

AND

CODE IN

RUBY