Ruby

O MELHOR AMIGO DO PROGRAMADOR

Gabriel Spelta

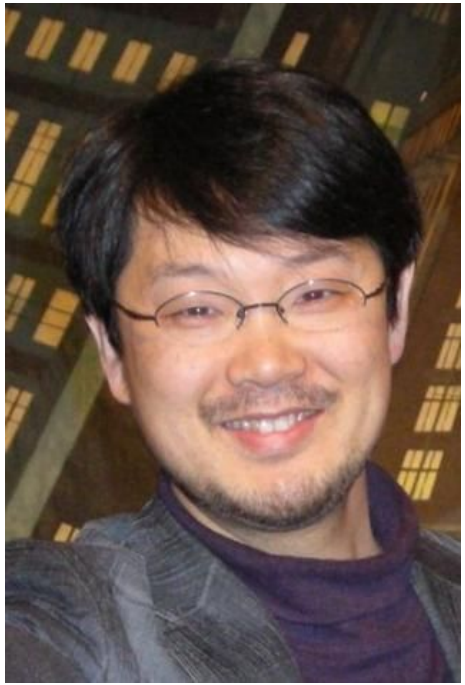
Vinicius Salles

Leonardo Lemos

Introdução:

“Eu queria uma linguagem de script que fosse mais poderosa do que Perl, e mais orientada a objetos do que Python. É por isso que eu decidi desenvolver minha própria linguagem.”

Ruby também sofreu influências de outras linguagens, como: **Smalltalk** (orientação a objetos), **Eiffel** (lambda calculus), **Lisp** (expressões regulares), etc.



Yukihiro Matsumoto

Introdução:

- ➔ Uma linguagem totalmente livre. Não possui custos e também pode ser usada para reutilização ou modificação.
- ➔ Mescla características de linguagem imperativa e funcional. Buscando o melhor das duas.
- ➔ Apesar de ter surgida em 1995, apenas começou a se tornar popular mundialmente a partir de 2006.
- ➔ Dúvida na escolha de nomes: “Coral” ou “Ruby”. Acabou por optar a segunda opção, pois era uma pedra zodiacal de um dos seus amigos.

Linha do Tempo:

- **1995:**
  - Criação do Ruby.
- **1999:**
  - Lista de discussão Ruby-Talk.
- **2000:**
  - Primeiro livro sobre Ruby em inglês.
- **2004:**
  - Surgimento do framework para web **Ruby on Rails**
- **2006:**
  - Linguagem do ano.

Ruby on Rails:

➔ O que é Ruby on Rails?

- “É um framework de desenvolvimento web escrito na linguagem Ruby. Designado para tornar a programação de aplicações web mais fácil, fazendo várias suposições sobre o que cada desenvolvedor precisa para começar.”
- Permite que se escreva menos código enquanto faz mais do que muitas outras linguagens e frameworks. Desenvolvedores que utilizam essa ferramenta há muito tempo, costumam dizer que ela torna a programação web mais divertida.

Como instalar?

- Compatível com Linux / Unix, OS X e Windows.
- É possível realizar download e obter instruções através do site:
  - https://www.ruby-lang.org/pt/downloads/
- Usuários das versões Debian ou Ubuntu e seus respectivos derivados, podem realizar o download através do comando:

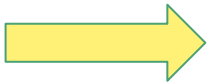
```
$ sudo apt-get install ruby-full
```

Então vamos lá!

Legibilidade:

- ➔ **Ruby é voltada para uma linguagem mais natural. Maioria dos seus comandos são em inglês.**
- ➔ **Muitas maneiras de se resolver um única maneira.**
- ➔ **Sintaxe semelhante a Python em alguns casos.**

Ex: 5.times{puts “Programando em Ruby”}



Programando em Ruby
Programando em Ruby
Programando em Ruby
Programando em Ruby
Programando em Ruby

Legibilidade:

➔ **A legibilidade também é influenciada pela forma em que as variáveis são declaradas. Dessa forma, é possível identificar com rapidez o papel que cada uma tem em seu contexto específico ou geral.**

```
local = "local" #variavel local.
@instancia = 25 #variavel de instância.
@@classe = " " #variavel de classe.
$Global = 3.14 #variavel global.
```

➔ **Blocos condicionais:**

```
x = 2
if (x % 2 == 0) then
    print ("X é par\n")
else
    print "X é impar!\n"
end
```

Ruby não possui goto nativamente, mas por meio de bibliotecas (ou gems) de terceiros, há um meio de ser “simulado”.

Redigibilidade:

- ➔ **Ruby possui uma preocupação maior com a redigibilidade do que com a legibilidade.**
- ➔ **Por exemplo, enquanto Python, linguagem na qual foi inspirada, possui 9 métodos para listas, Ruby possui cerca de 79. Isso torna a resolução de problemas muito mais fácil, porém prejudica a legibilidade por terceiros.**
- ➔ **Uma entrevista do criador revelou que ele prefere dar opções ao programador, dar ferramentas para utilizar da maneira que achar necessário. Sendo ainda, possível programar ao estilo de Python em vez da forma criada primeiramente para Ruby.**

Confiabilidade:

- **Ruby possui variáveis de tipos diferentes, porém, todas são classes e não é necessário declarar tipos.**
- **Ruby possui tratamento de exceções semelhante ao de Java e Python, o que é um ponto positivo para a confiabilidade e facilita bastante o tratamento de erros.**

  Ruby possui palavras-chaves para o tratamento de exceções: begin, raise, rescue, ensure e retry.

Facilidade de Aprendizado:

- ➔ **A sintaxe de Ruby é bastante influenciada pela sintaxe da linguagem Eiffel. E, do ponto de vista do aprendizado, a sintaxe é muito simples. Chega a ser intuitiva em alguns casos.**
- ➔ **O surgimento do primeiro livro em inglês (uma linguagem global), Programming Ruby, facilitou o aprendizado e ajudou a popularizar a linguagem no ocidente.**
- ➔ **No próprio site da linguagem, há um tutorial onde se pode “aprender” Ruby em apenas vinte minutos. Mostrando suas funcionalidades básicas e sintaxe.**

Facilidade de Aprendizado:

- ➔ **Ponto negativo para a linguagem é o excesso de métodos de se escrever uma mesma coisa.**
- ➔ **É bom conhecer um pouco de orientação a objetos, caso contrário, pode ser mais trabalhoso para compreender as funcionalidades de Ruby.**

Ortogonalidade:

- **Criador da linguagem diz que ortogonalidade pode ser algo ruim. Devido a isso, foi-se aberto mão de parte da ortogonalidade em função de uma linguagem mais simples.**

Modificabilidade:

- **Constantes:**
  - **Para se declarar uma variável como constante, basta declarar com a primeira letra maiúscula. Exemplo:**

```
Pi = 3.1415 #constante declarada com nome Pi.
Pi  = 2         #é mutável, porém causa um warning.
```

Reusabilidade:

- **Pelo fato de Ruby ter sua parcela de orientação à objetos, possui todas as suas variáveis sendo classes, o código nessa linguagem é altamente reutilizável.**
- **Ruby também possui um recurso chamado gemas. As gemas (que podem ser chamadas de pacotes ou aplicações) são partes de programas que auxiliam na hora da geração do código. As gemas podem ser encontradas através do gerenciador RubyGems.**

```
def pessoa (nome)
    "ola, eu sou " + nome
end
puts pessoa("Fulano")
```

ola, eu sou Fulano

Implementação e Paradigma:

- ➔ **Ruby é uma linguagem interpretada, ou seja, não existe um processo de compilação para um binário executável, como acontece na linguagem C, por exemplo.**
- ➔ **Possui um paradigma de Orientação à Objetos. Tudo é classe. E um enfoque em paradigma funcional.**

Portabilidade:

- ➔ **Por ser interpretada, isso já garante uma alta portabilidade. Como já citado, pode ser utilizada nos principais sistemas operacionais.**
- ➔ **Implementa threads totalmente em nível de usuário, o que o torna independente do sistema operacional.**

Sintaxe:

➔ **Operadores:**

+	(Adição)	>=	(Maior igual)	A==B ? A : B	(Ternário)
-	(Subtração)	<=	(Subtração)	..	(range inclusivo)
*	(Multiplicação)	and / &&	(“e” condicional)	…	(range exclusivo)
/	(Divisão)	or / ||	(“ou” condicional)	||=	(verifica nil e atribua valor)
%	(Módulo)	not	(“negação” condicional)		
**	(Expoência)	&	(“e” bit-a-bit)		
==	(Igual)	|	(“ou” bit-a-bit)		
!=	(Diferente)	!	(“negação bit-a-bit)		
>	(Maior)	~	(complemento de 1)		
<	(Menor)	>> / <<	(deslocamento esq/dir)		

Sintaxe:

➔ **Palavras reservadas em Ruby:**

alias - and - BEGIN - begin - break - case - class - def

defined - do - else - elsif - END - end - ensure - false

for - if - in - module - next - nil - not - or - redo – rescue

retry - return - self - super - then - true - undef – unless

until - when - while - yield

Sintaxe:

➔ **Conversão:**

```
x = 1.5
puts x.class #Float

y = x.to_i
puts y.class #Fixnum

s = y.to_s
puts s.class #String
```

➔ **Ternário:**

```
x = 2
(x == 2) ? (x += 1) : (x-= 1)
puts (x) #3 (x = x + 1)
```

Escopo, Blocos e Declarações:

- **Escopo:**
  - **Ruby possui um escopo estático, ou seja, suas variáveis têm seus escopos determinados antes da execução do programa.**
- **Blocos:**
  - **Ruby possui uma implementação do tipo blocos aninhados.**
- **Declarações:**
  - **Definições:**
    - x = 2; y=3
  - **Declaração de variáveis:**
    - classe = Classe.new
  - **Declaração de tipos:**
    - class Class

      @var

      end

Tipos de Dados:

- **TIpagem:**
  - **Ruby possui uma tipagem:**
    - **Dinâmica**
    - **Forte**
    - **Implícita**

```
x = 100

7.times {puts "#{x.class} -> #{x}"; x *= 1000}

puts x + "ola" #TypeError: String can't be coerced into Fixnum
```

```
Fixnum -> 100
Fixnum -> 100000
Fixnum -> 100000000
Fixnum -> 100000000000
Fixnum -> 100000000000000
Fixnum -> 10000000000000000
Bignum -> 100000000000000000000
```

Tipos de Dados:

- **Object:** todas as classes herdam desta classe.
  - **Numeric:** classe para todos os números.
    - **Integer:** classe para os números inteiros.
      - **Fixnum:** classe para inteiros limite finito.
      - **Bignum:** classe para inteiros limite quase infinito (depende da memória)
    - **Float:** classe para todos os números reais.
  - **String:** classe para textos, delimitadas por aspas duplas (“) ou simples (‘).
  - **Symbol:** parecidas com strings, porém não possuem tanta flexibilidade. São muito utilizadas em Hash’s.
  - **Array:** classe para trabalhar com Arrays
  - **Hash:** classe para trabalhar com Hash Maps.

Tipos de Dados:

- **Strings**:
  - **Utiliza o padrão encoding de caracteres é USA-ASCII. Porém, caso seja necessário, pode-se alterar utilizando o comando: #coding: utf-8 (por exemplo).**
  - **Existem diversas maneiras de se criar Strings em Ruby (como todas as outras coisas que se podem fazer de diversas maneiras):**
    - **texto = “oi”; texto = ‘oi’**
    - **texto = String.new(“oi”)**
    - **texto = %~oi~ #pode ser colocado qualquer símbolo ([{ ^ ,)**

Tipos de Dados:

- **Symbols:**
  - **Um símbolo (Symbol) se parece com um nome de variável mas é prefixado com dois pontos (:). Você não precisa pré-declarar um símbolo e garante-se que eles sejam únicos.**
  - **São bastantes usados em Hash's como identificadores para chaves.**
    - **teste = :isso_e_um_symbol**
    - **if teste == :isso_e_um_symbol**

Tipos de Dados:

- **Array:**
  - **Armazena quaisquer objetos indexando por inteiros. Sendo o primeiro valor 0. Pode-se acessar de modo circular através de índices negativos.**
  - **É alocado dinamicamente na memória.**
  - **Possui várias funções padrões: take, drop, push (<<) , pop, delete, etc.**

```
irb(main):001:0> array = [1,"oi",false,2.5]
=> [1, "oi", false, 2.5]
irb(main):002:0> array << "inserindo"
=> [1, "oi", false, 2.5, "inserindo"]
irb(main):003:0> array.pop
=> "inserindo"
irb(main):004:0> array.delete(false)
=> false
irb(main):005:0> array
=> [1, "oi", 2.5]
```

Tipos de Dados:

- **Hash:**
  - **Uma estrutura semelhante ao array, porém, seus elementos estão associados à um objeto que funciona como uma chave.**

nome_da_variavel = {objeto_chave1 => valor1, objeto_chave2 => valor2}

```
irb(main):001:0> hash = {1 => "ola", true => "verdadeiro", 2.5 => false}
=> {1=>"ola", true=>"verdadeiro", 2.5=>false}
irb(main):002:0> hash[true]
=> "verdadeiro"
irb(main):003:0> hash.has_key?(1)
=> true
irb(main):004:0> hash[:inserindo] = 100
=> 100
irb(main):005:0> hash
=> {1=>"ola", true=>"verdadeiro", 2.5=>false, :inserindo=>100}
```

Tipos de Dados:

- **Classe:**
  - **Como tudo são classes, nada mais justo do que poder criar classes do que desejar.**

```
Class Pessoa
    attr_reader :nome
    attr_writer :nome
    #attr_accessor :nome

    def initialize (nome)
        @nome = nome
    end

    def to_s
        puts @nome
    end
end
pessoa = Pessoa.new("joao")
puts pessoa.nome  # => joao
pessoa.nome = "outro joao"
puts pessoa.nome # => outro joao
```

Variáveis:

- Em Ruby, existem 4 tipos de variáveis diferentes:
  - Variáveis Local: tem um escopo local;

    var = 10
  - Variáveis de Instância: é referenciada por uma instância, por isso pertence a um objeto;

    @var= 10
  - Variáveis de Classes: é partilhada por todos os objetos da classe;

    @@var = 10
  - Variáveis Globais: pode ser acessada globalmente por qualquer estrutura;

    $var = 10

Variáveis:

➔ Propriedades – Nome e Valor:

- Declaração de variáveis é uma associação entre um nome e um valor;
- A atribuição é realizado com o sinal **=**;

```
nota_do_seminario = 10
_nome = ”Zé Colmeia”
```

➔ Propriedades – Tipo:

- Os tipos das variáveis são inferidas automaticamente durante a execução do código;

```
irb(main):001:0> nota_do_seminario.class
=> Fixnum
irb(main):002:0> _nome
=> String
```

Variáveis:

- ➔ Propriedades – Endereço:
- ◆ O endereço de memória não é acessível, ou seja, não há aritmética de ponteiros;
- ◆ Apesar dos objetos contarem com um identificador, eles não são o endereço de memória;

```
irb(main):001:0> _nome.object_id.to_s(16)
=> “1bd7e7c”
```

- ➔ Propriedades – Tempo de Vida e Escopo:
- ◆ **Variáveis locais:** ambos limitado ao método referenciado;
- ◆ **Variáveis de instância:** possuem escopo local e tempo de vida estático;
- ◆ **Variáveis de classes:** possuem escopo local e tempo de vida estático;
- ◆ **Variáveis globais:** ambos acompanham o processo

Constantes:

➔ São basicamente variáveis que possuem o mesmo valor durante a execução do programa;

PI = 3.141592

➔ Ao contrário da maioria das linguagens as constantes são mutáveis, ou seja, os valores pode ser alterados;

```
irb(main):001:0> PI = 3.1415
=> 3.1415
irb(main):002:0> PI = 3.141592
(irb):2: warning: already initialized constant PI
(irb):1: warning: previous definition of PI was here
⇒    3.141592
```

Serialização e Deserialização:

➔ Refere-se a variável transiente que é convertida de sua representação na memória primária para uma sequência de bytes na memória secundária, e vice-e-versa;

```
irb(main):001:0> hash = {nome: “Ze Colmeia”}
=> {:nome=>“Ze Colmeia”}
irb(main):002:0> hash_serializado = Marshal.dump(hash)
=> “\x04\b{\x06:\tnomel\”\x0FZe Colmeia\x06:\rencoding\”nCP850”
irb(main):003:0> hash_deserializado = Marshal.load(hash_serializado)
=> {:nome=>“Ze Colmeia”}
```

Expressões:

- ➔ Uma expressão é uma frase no programa que precisa ser avaliada e produz como resultado um valor:
  - ◆ Operador + Operandos => Resultado

- ➔ Chamada de funções/métodos também são consideradas expressões:
  - ◆ Nome da Função + Parâmentros => Retorno da Função

- ➔ Quanto a tipo de expressões, tem-se:
  - ◆ Aritméticas, relacionais, binárias, booleanas, condicionais;
  - ◆ Unárias, binárias e ternárias;
  - ◆ Prefixada e Infixada;

Expressões:

- Aritméticas:

Operador	O que faz?	Exemplo	Método?
+	Adiciona valores da esquerda com o da direita	`a + b` irá retornar 30	Sim
-	Subtrai os valores da esquerda com o da direita	`b - a` irá retornar 10	Sim
*	Multiplica os valores	`a * b` irá retornar 200	Sim
/	Divide o valor da esquerda com o da direita	`b / a` irá retornar 2	Sim
%	Divide o valor da esquerda com o da direita e retorna a sobra da divisão	`b % a` irá retornar 0	Sim

- Lógica:

Operador	O que faz?	Exemplo	Método?
and ou &&	Se ambas condições forem verdadeiras, então é retornado `true`	`(a && b)` é verdadeiro	Não
or ou double pipe	Se algumas das condições forem diferente de zero, o retorno é `true`	`(a or b)` é verdadeiro	Não
not ou !	Inverte o estado da lógica do operando	`!(a && b)` é falso	Não

Expressões

- Binário: **(x = 18).to_s(2) # =>”10010” (y = 20).to_s(2) # =>”10100”**

Operador	O que faz?	Exemplo	Método?
&	Operador AND	`(x & y).to_s(2)` irá retornar "10000"	Sim
pipe	Operador OR	`(x pipe y).to_s(2)` irá retornar "10110"	Sim
^	Operador XOR	`(x ^ y).to_s(2)` irá retornar "110" (zeros à esquerda são omitidos)	Sim
~	Operador NOT	`(~x).to_s(2)` irá retornar "-10011"	Sim
<<	Operador com desvio a esquerda	`(x << 2).to_s(2)` irá retornar "1001000"	Sim
>>	Operador com desvio a direita	`(x >> 2).to_s(2)` irá retornar "100"	Sim

- Lógico:

Operador	O que faz?	Exemplo	Método?
? :	Expressão condicional	Se a condição é verdadeira? então o valor é `x` caso contrário o valor é `y`	Não

Expressões:

- Booleano/Comparativo:

Operador	O que faz?	Exemplo	Método?
==	Verifica se o valor dos dois operandos são iguais ou não, se sim, então a condição se torna verdadeira.	`a == b` não é verdadeiro	Sim
!=	Verifica se o valor dos dois operandos são diferentes ou não, se sim, então a condição se torna verdadeira.	`a != b` é verdadeiro	Sim
>	verifica se o valor da esquerda é maior que o da direita, se sim retorna verdadeiro	`a > b` não é verdadeiro	Sim
<	verifica se o valor da esquerda é menor que o da direita, se sim retorna verdadeiro	`a > b` é verdadeiro	Sim
>=	verifica se o valor da esquerda é maior ou igual ao da direita, se sim retorna verdadeiro	`a >= b` não é verdadeiro	Sim
<=	verifica se o valor da esquerda é menor ou igual ao da direita, se sim retorna verdadeiro	`a <= b` é verdadeiro	Sim

Expressões:

- Booleano/Comparativo:

Operador	O que faz?	Exemplo	Método?
<=>	Operador de comparação combinada. Retorna `0` se primeiro operando é igual ao segundo, retorna 1 se o primeiro operando for maior que o segundo e retorna `-1` se o primeiro operando for menor que o segundo.	`a <=> b` retorna -1	Sim
===	Usado para testar a igualdade dentro de uma cláusula que contenha instruções.	`(1...10) === 5` é verdadeiro	Sim
.eql?	Verdadeiro se o receptor e o argumento têm o mesmo tipo e valor.	`1 == 1.0` retorna verdadeiro, mas em `1.eql?(1.0)` não é verdadeiro	Sim
.equal?	Verdadeiro se o receptor e o argumento tem a mesma identificação do objeto.	se `foo_obj` é duplicado para `bar_obj` então `foo_obj == bar_obj` é verdade, `a.equal? bar_obj` é falso, mas `a.equal? foo_obj` é verdade.	Sim

Expressões:

➔ Métodos como expressões:

```
def boas_vindas (nome)
        “Olá, ” + nome + “. Seja bem-vindo!”
end

puts boas_vindas (“Zé Colmeia”)

# =>Olá, Zé Colmeia. Seja bem-vindo!
```

```
def boas_vindas (nome)
        return “Olá, ” + nome + “. Seja bem-vindo!”
end

puts boas_vindas (“Zé Colmeia”)

# =>Olá, Zé Colmeia. Seja bem-vindo!
```

Expressões:

➔ Avaliação de curto-circuito:

```
def a
        puts “A foi calculada”
        return true
end

def b
        puts “B foi calculada”
        return false
end

puts a || b
puts “-----”
puts b && a
```

```
A foi calculada
true
-----
B foi calculada
false
```

Comandos:

➔ Atribuição:

Tem-se presente atribuição simples, múltipla, condicional, composta e por expressão.

```
irb(main):001:0> a = 2+3*2
=> 8
irb(main):002:0> b = c = 0
=> 0
irb(main):003:0> a<b : b=2 ? c=3
=> 3
irb(main):00a:0> if (c+=3)>5; puts “Deu bom”; end
Deu bom
=> nil
```

Comandos:

➔ Sequenciais:

Há comandos sequenciais e bloco de comandos. Usa como delimitador de blocos a palavra **end**, sendo que o bloco inicia-se com a palavra do método a ser implementado ou que foi chamado, ou das estruturas condicionais (if, for, while…).

```
if x<y
        n = 1
        n *= 3
        if n<5
                n = 10
                m = n*2
        end
end
```

Comandos:

➔ Condicional:

Há seleção de caminho condicionado, caminho duplo e múltiplos caminhos. Possui também marcadores bem integrados (end), que reduz os problemas de comandos condicionais.

```
puts case a
when 1..5
        “Está entre 1 e 5”
when 6
        “É o número 6”
when String
        “Você me passou uma string”
else
        “Você me passou #{a} -- Não sei o que fazer
end
```

Comandos:

➔ Iterativo:

Há comandos com número indefinido de repetições, tanto pré-teste (while), quanto pós-teste (do-while), e comandos com número definido de repetições (for);

```
loop do
        a = a+1
        break if a == 10
end
```

```
while not a == 10
        a = a + 1
end
```

```
for a in 2...10
        a +=1
end
```

Comandos:

➔ Desvios Incondicionais:

Ruby não possui desvio incondicional (goto), mas existem gems que possuem esse método implementado.

➔ Escapes:

Break, next e *return* funcionam como *escapes*, apesar da linguagem não possuir desvio incondicional.

Modularização:

- ➔ Ruby é uma linguagem modularizada por sua essência, já que se trata de uma linguagem orientada a objeto;

- ➔ Assim como Java, tem para cada classe, um arquivo .rb a ser implementado, matendo a modularização;

Modularização:

➔ Abstração de Processos:

- Subprogramas: Essência da O.O., segmenta o programa tornando-o mais fácil de implementar, aumentando assim a reusabilidade, legibilidade, reutilização, e facilita na depuração e manutenção;

➔ Parâmetros:

- Aumenta a redigibilidade e legibilidade, pela escrita na implementação, e aumenta a confiabilidade, pois diminui o uso de variáveis globais;
- Aceita valores default nos parâmetros, possui passagem por cópia da referência e, se tratando de objetos, por ter passagem bidirecional;

Modularização:

➔ Parâmetros:

```
class Cilindro
        def initialize (a, b)
                a = a
                b = b
        end
        def v
                (a**2)* PI*b
        end
end
```

```
class Cilindro
        def initialize (raio, altura)
                @raio = raio
                @altura = altura
        end
        def volume
                (@raio**2)* PI*@altura
        end
end
```

Modularização:

➔ Abstração de Dados:

- Há uma forte verificação de dados do interpretador (linguagem fortemente tipada), e possui tipos de dados simples bem definido, além de cuidar disso para o programador (inferência dinâmica de tipo);
- Por ser O.O., Ruby incentiva o uso de Tipo Abstrato de Dados (TAD);
- Das quatro diferentes operações principais em TAD – construtora, consultora, atualizadora e destrutora – a linguagem apresenta a solução de duas.

```
def initialize (a, b, ...)
...
end
```

Garbage Collector

Polimorfismo

Polimorfismo

- **Coerção:**
  - **Ruby possui polimorfismo de coerção, exemplos:**

    5 + 2.5 #saída> 7.5 int -> float
    4 * 2.7 #saída> 10.8 int -> float

- **Sobrescrita:**
  - **Como as classes são abertas no Ruby, e os operadores são métodos, esses podem ter seus comportamentos sobrescritos.**

```
class Fixnum
    def + (outro)
        self - outro
    end
end

puts 2+2 #saída = 0
```

Polimorfismo

- **Paramétrico:**
  - Os métodos de Ruby são paramétricos por natureza, vide que não é necessário declarar tipo dos parâmetros.

```
def soma (a, b)
        puts a + b
end
```

- **Inclusão:**
  - Pelas heranças de classe de Ruby, é possível ter esse tipo de polimorfismo, porém é necessário cuidado ao usar essa técnica, pois não há encapsulamento **entre** objetos.

Amarração

- Por ser uma linguagem interpretada, Ruby faz uso de amarração tardia.

Classes e métodos abstratos

- Ruby não oferece suporte a classes abstratas, mas oferece suporte a métodos abstratos.

```
class ClasseAbstrata

       def metodo_um; raise "ResponsabilidadeDaSubclasse" ;
end

       def metodo_dois; raise "ResponsabilidadeDaSubclasse"
; end

end
```

Manipulação de exceções

Lançamento de exceções

```
begin
    # faz alguma coisa
    raise 'Ocorreu um erro.'
    rescue => e
    puts 'Eu fui resgatado.'
    puts e.message
    puts e.backtrace.inspect
end
```

raise == fail

```ruby
begin
    # faz alguma coisa
    fail 'Ocorreu um erro.'
    rescue => e
    puts 'Eu fui resgatado.'
    puts e.message
    puts e.backtrace.inspect
end
```

“Eu quase sempre uso a palavra-chave “fail”... A única vez em que uso “raise” é quando eu tenho que pegar uma exceção e re-lançá-la, porque aqui eu não estou falhando, mas explicitamente e propositalmente lançando uma exceção.”

– Jim Weirich, Autor do Rake

Assinatura do método “raise”

raise (classe_de_excessao_ou_objeto, mensagem, backtrace)

raise

```
raise

# é equivalente a:

raise RuntimeError
```

raise (string)

```
raise ‘Ocorreu um erro.’

# é equivalente a:

raise RuntimeError, ‘Ocorreu um erro.’
```

raise (excessao, string)

```
raise RuntimeError, ‘Ocorreu um erro.’

# é equivalente a:

raise RuntimeError.new(‘Ocorreu um erro.’)
```

Variável global *$!*

- Mantém a referência para a exceção ativa atualmente;
- Resetada para *nil* quando a exceção for resgatada.

rescue

```
rescue AlgumErro => e
     # ...
end
```

Múltiplas classes ou módulos

```
rescue AlgumErro, OutroErro => e
     # ...
end
```

Empilhar rescues (a ordem importa)

```
rescue AlgumErro => e
     # ...
rescue AlgumOutroErro => e
     # ...
end
```

```
rescue
     # ...
end
# é equivalente à:
rescue StandardError
     # ...
end
```

Hierarquia de Excessões de Ruby

- Exception
  - NoMemoryError
  - LoadError
  - SyntaxError
  - StandardError -- Padrão para rescue
    - ArgumentError
    - IOError
    - NoMethodError
    - ...

Evite capturar a classe Exception

```
rescue Exception => e
     # ...
end
```

Utilizando suppress com exceções

```
suppress(IOError, SystemCallError) do
     open("ARQUIVO_NAO_EXISTENTE")
end

puts 'Este código é executado.'
```

ensure

```
begin
     # faz alguma coisa
     raise 'Aconteceu um erro.'
rescue => e
     puts 'Eu fui resgatado.'
ensure
     puts 'Este código sempre será executado.'
end
```

retry

```
tentativas = 0
begin
     tentativas += 1
     puts "Tentando #{tentativas}..."
     raise "Não funcionou :("
rescue
     retry if tentativas < 3
     puts "Eu desisto!"
end
```

else

```
begin
     yield
rescue
     puts "Aconteceu um erro..."
else
     puts "Ah, não, deu tudo certo! :D"
ensure
     puts "Sempre executado!"
end
```

Concorrência

Threads

```
# Thread 1 está rodando aqui


Thread.new {
  # Thread 2 está rodando aqui
}


# Thread 1 roda este código
```

Threads - exemplo

```ruby
#!/usr/bin/ruby

def func1
   i=0
   while i<=2
      puts "func1 em: #{Time.now}"
      sleep(2)
      i=i+1
   end
end

def func2
   j=0
   while j<=2
      puts "func2 em: #{Time.now}"
      sleep(1)
      j=j+1
   end
end

puts "Começou em #{Time.now}"
t1=Thread.new{func1()}
t2=Thread.new{func2()}
t1.join
t2.join
puts "Terminou em #{Time.now}"
```

```
Começou em Wed May 14 08:21:54 -0700 2008
func1 em: Wed May 14 08:21:54 -0700 2008
func2 em: Wed May 14 08:21:54 -0700 2008
func2 em: Wed May 14 08:21:55 -0700 2008
func1 em: Wed May 14 08:21:56 -0700 2008
func2 em: Wed May 14 08:21:56 -0700 2008
func1 em: Wed May 14 08:21:58 -0700 2008
Terminou em Wed May 14 08:22:00 -0700 2008
```

Ciclo de vida de uma Thread

- Threads são criadas usando Thread.new, mas pode ser criada usando Thread.fork e Thread.start
- Não existe a necessidade de se iniciar uma Thread logo após sua criação
- Existem métodos que podem manipular uma Thread
- Thread.current retorna o objeto da thread atual; Thread.main retorna a thread principal do programa em Ruby
- Você pode chamar uma Thread de volta utilizando Thread.join

Threads e exceções

- Qualquer exceção não tratada matará uma thread mas fará todo o resto rodar normalmente
- Essa condição é tratada pelo atributo Thread.abort_on_exception, que por padrão assume o valor false
- Se você deseja que todo o programa aborte neste caso, utilize Thread.abort_on_exception = true

Threads e exceções - exemplo

```
t = Thread.new { ... }
t.abort_on_exception = true
```

Variáveis de Thread

- Threads podem acessar qualquer variável que exista no escopo em que forem criadas
- A classe Thread permite que sejam criadas e acessadas variáveis locais no escopo da Thread

```
#!/usr/bin/ruby

count = 0
arr = []

10.times do |i|
   arr[i] = Thread.new {
      sleep(rand(0)/10.0)
      Thread.current["mycount"] = count
      count += 1
   }
end

arr.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

```
8, 0, 3, 7, 2, 1, 6, 5, 4, 9, count = 10
```

Prioridades de Thread

- Threads com maior prioridade são executadas antes das de menor prioridade
- Uma Thread só terá tempo de CPU se uma Thread se não houver uma Thread de maior prioridade na fila de execução aguardando por tempo de CPU
- Você pode mudar a prioridade da Thread utilizando Thread.priority =

Prioridades de Thread

- Threads com maior prioridade são executadas antes das de menor prioridade
- Uma Thread só terá tempo de CPU se uma Thread se não houver uma Thread de maior prioridade na fila de execução aguardando por tempo de CPU
- Você pode mudar a prioridade da Thread utilizando Thread.priority =

Exclusão de Thread

- Para garantir que duas ou mais Threads que compartilham recursos não acessem e utilizem dados inconsistentes destes recursos, utilizamos a classe Mutex
- Mutex implementa uma trava para acesso mútuo para recursos compartilhados
- Apenas uma Thread pode possuir a trava por vez

Exclusão de Threads - exemplo

```ruby
#!/usr/bin/ruby
require 'thread'

count1 = count2 = 0
diferenca = 0
contador = Thread.new do
   loop do
      cont1 += 1
      cont2 += 1
   end
end
espiao = Thread.new do
   loop do
      diferenca += (cont1 - cont2).abs
   end
end
sleep 1
puts "cont1 :  #{cont1}"
puts "cont2 :  #{cont2}"
puts "diferença : #{diferenca}"
```

```
cont1 :  1583766
cont2 :  1583766
diferença : 637992
```

```ruby
#!/usr/bin/ruby
require 'thread'
mutex = Mutex.new

cont1 = cont2 = 0
diferenca = 0
contador = Thread.new do
   loop do
      mutex.synchronize do
         cont1 += 1
         cont2 += 1
      end
    end
end
espiao = Thread.new do
   loop do
       mutex.synchronize do
          difference += (cont1 - cont2).abs
       end
   end
end
sleep 1
mutex.lock
puts "cont1 :  #{cont1}"
puts "cont2 :  #{cont2}"
puts "diferença : #{diferenca}"
```

```
cont1 :  696591
cont2 :  696591
diferença : 0
```

Avaliação da linguagem

Critérios Gerais	C	Java	Ruby
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Parcial
Eficiência	SIm	Parcial	Não
Portabilidade	Não	SIm	Sim
Método de Projeto	Estruturado	OO	OO + Funcional
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Parcial	Sim	Sim
Integração	Sim	Parcial	Parcial
Custo	Depende	Depende	Depende

Avaliação da linguagem

Critérios Gerais	C	Java	Ruby
Persistência de dados	Biblioteca Funções	Biblioteca classes, serialização, JDBC	Coleções, serialização, BD, gems.
Passagem de parâmetros	Lista variável e por valor	Lista variável, por valor e por cópia ref	Lista variável e por cópia da referência.
Encapsulamento e proteção	Parcial	Sim	Parcial
Verificação TIpos	Estática	Estática / Dinâmica	Dinâmica
Polimorfismo	Coerção; Sobrecarga	Todos	Quase todos
Exceções	Não	Sim	SIm

Avaliação da linguagem

Critérios Gerais	C	Java	Ruby
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
TIpo primitivos e Compostos	Sim	Sim	Parcial
Ger. Memória	Programador	Sistema	Sistema

Referências

- https://pt.wikipedia.org/wiki/Ruby_(linguagem_de_programação)
- https://sites.google.com/a/cin.ufpe.br/floss/linhadotempo
- http://guru-sp.github.io/tutorial_ruby/
- Ruby - Aprenda a programar na linguagem mais divertida - Casa do Código (Lucas Souza)
- https://www.ruby-lang.org/pt/