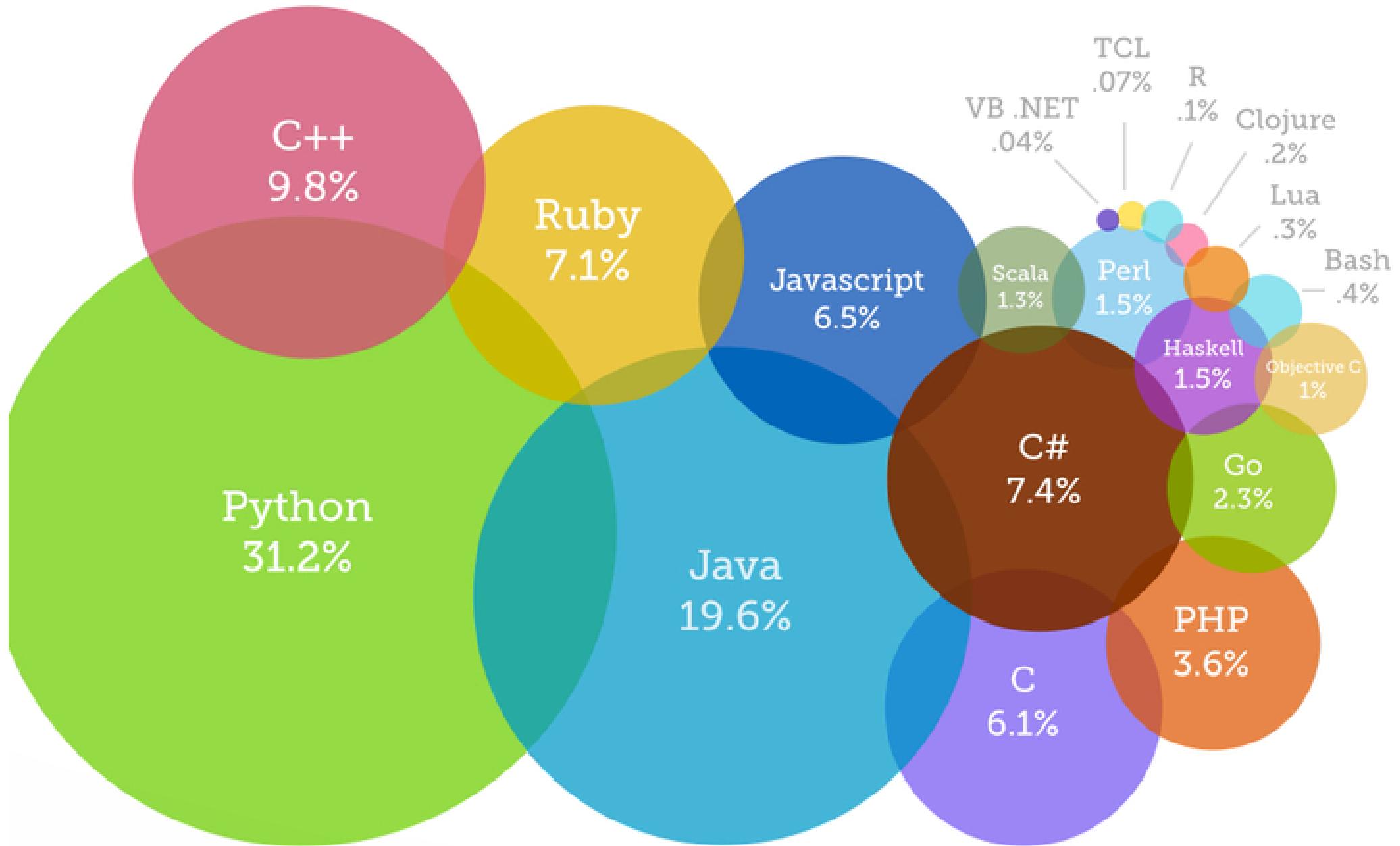


Linguagem de Programação para Web

Ruby
Prof. Tales Bitelo Viegas

Most Popular Coding Languages of 2015



Por que Ruby?

- ▶ Comunidade

Existe uma grande comunidade de Ruby na internet, é muito fácil achar informações, exemplos de código, projetos open-source e mais.

<http://www.rubygems.org>



Por que Ruby?

- ▶ A sintaxe fácil e a filosofia da linguagem favorecem um desenvolvimento mais ágil
- ▶ O framework Ruby On Rails, que ganha cada vez mais espaço atualmente, é um exemplo dos resultados que se pode obter.
 - Convention over Configuration
 - Don't Repeat Yourself



Começando

- ▶ É bom saber que a instalação de Ruby inclui o *irb* (Interactive Ruby Shell), um interpretador que nos permite escrever código e receber imediatamente o retorno.
- ▶ Também vem incluso o RI, que fornece a documentação.



Por que Ruby?

- ▶ Ruby é dinâmica

```
01. a = "variavel"  
02. puts a.class #String  
03. a = 2  
04. a += 3  
05. puts a.class #Fixnum
```

A tipagem das variáveis é dinâmica, determinada em tempo de execução pelo interpretador

Por que Ruby?

- ▶ Ruby é mais dinâmica

```
01. class String
02. #adicionando um metodo à classe string:
03. #retorna o primeiro char de uma palavra
04.   def primeiro
05.     return "" << self[0]
06.   end
07.
08. end
09.
10. puts "Ruby".primeiro #imprime "R"
```

Até mesmo as classes do core do Ruby são abertas e podemos modificá-las em tempo de execução

Por que Ruby?

- ▶ DRY – Don't Repeat Yourself
- ▶ Ruby favorece código pequeno e elegante

```
01. def fat n  
02.   (2..n).inject do |x,y|  
03.     x*y  
04.   end  
05. end
```

Começando

- ▶ Hello World

```
01. | print "Hello World"
```

- ▶ Observamos aqui: usamos o método print para imprimir a string “Hello World” no console. Notem que a delimitação dos comandos é uma nova linha

Começando

```
01. #note que assim tb funciona
02. print 'Hello World'
03. #vemos que foi impresso Hello WorldHello World
04. #tentamos
05. print "Hello World\n"
06. #mas podemos esquecer disso usando
07. puts "Hello World"
08. #tambem podemos ver que o exemplo a seguir
09. #nao funciona como o esperado
10. print '\nHello World\n'
```

Escopo das Variáveis

- ▶ Como em Ruby só declaramos as variáveis no momento estritamente necessário, precisamos ficar atentos para o escopo das mesmas.
- ▶ Temos os seguintes escopos de variáveis:
 - Local – declaradas normalmente com letra minúscula ou começando com o caracter “_”
 - Global – declaradas com “\$” na frente do nome
 - Classe – declaradas com “@” na frente do nome



Escopo das Variáveis

► Exemplificando

```
01. #Aqui veremos os dois primeiros escopos.  
02. a = 15# local para esse escopo  
03.  
04. 3.times { |_Var_local| puts _Var_local }  
05. puts _Var_local #aqui teríamos um erro  
06.  
07. 5.times do |c|  
08.   $evil = "Muahuahuahuahauua"  
09.   puts $evil  
10. end  
11.  
12. puts $evil #sem problemas
```

Operadores

- ▶ Em Ruby temos operadores aritméticos e booleanos muito semelhantes aos de outras linguagens.
- ▶ Observem a tabela a seguir:



Operadores

Operador	Descrição
+ -	Soma e subtração
* / %	Multiplicação, divisão e módulo da divisão
**	Exponenciação
<= < > >= == !=	Operadores de comparação
&& and	“E” lógico
or	“OU” lógico
! not	Negação

Constantes

```
01. =begin
02. Também é possível comentar blocos do código assim.
03. É importante aprender a declarar constantes, elas
04. começam com letra maiúscula (não precisa ser
05. o nome inteiro).
06. =end
07.
08. NUMERO_DE_EULER = 2.7182818
09.
10. #Constantes também possuem escopo
11. class Evil
12. LAUGTHER = "Muhaahuahuahuahuahu"
13. end
14.
15. puts Evil::LAUGTHER
```

Constantes

- ▶ Um aspecto “interessante” em Ruby é que nós podemos mudar o valor de constantes sem culpa.
- ▶ O interpretador reclama mas consente

```
01. | Evil::LAUGHTER = "hihihihihihi"
02. | NUMERO_DE_EULER = 3.14159265 # :0
```

Tipagem Dinâmica e Forte

- ▶ Ruby é dinâmica e estaticamente tipada. Isso quer dizer que as variáveis são do tipo de objeto que elas referenciam em determinado momento do programa.

```
01.  resposta = 42
02.  puts resposta.class
03.  resposta = "quarenta e dois"
04.  puts resposta.class
```

- ▶ Porém ela também é forte, ou seja, não podemos fazer algumas “gracinhas”, como demonstrado a seguir...

Tipagem Dinâmica e Forte

```
01. | resposta += 2
02. | #isso vai nos gerar:
03. | #TypeError: can't convert Fixnum into String
```

O operador `+=` não está definido para a soma de um Fixnum com uma String, portanto foi gerado um erro.

Ruby evita fazer conversões implícitas.

Métodos

- ▶ Em linguagens como C ou C++ temos funções, mas em Ruby nosso trabalho será feito por métodos, o que faz sentido já que tudo é um objeto.

```
01. def dizer_ola nome
02.   #interpolação de strings
03.   puts "Olá #{nome}!"
04. end
05.
06. def dizer_tchau nome
07.   puts "Tchau #{nome}."
08. end
```

Métodos

```
01. def dialogo nome
02.     dizer_ola(nome)
03.     puts "Bla bla bla..."
04.     dizer_tchau(nome)
05. end
06.
07. meu_nome = 'Lucas'
08. nome_dele = 'Bruno'
09.
10. dialogo(meu_nome)
11. dialogo(nome_dele)
```

Cada método é um novo escopo, portanto podemos ter variáveis locais. Não é possível declarar constantes dentro de métodos.

Métodos

- ▶ O último valor executado no método será retornado.

```
01. def soma(x,y)
02.     return x + y
03. end
04. #mesma coisa que fazer
05. def soma2(x,y)
06.     x + y
07. end
```

Métodos

- ▶ Em Ruby é possível utilizar os caracteres ‘?’ e ‘!’ no nome dos métodos.
- ▶ Como convenção informal, podemos usar esses caracteres para deixar nossos métodos mais intuitivos.
- ▶ ? em métodos query (fazem uma pergunta)
- ▶ ! em métodos destrutivos (modificam o objeto)



Métodos

- ▶ Métodos query, uma pergunta que geralmente é respondida com um boolean

```
01. #exemplo de métodos query
02. array = [1,2,3,4,5,6] ; array.include?(6)
03.
04. # uma função que verifica se uma string contém um inteiro
05. def is_inteiro? string
06.   string.to_i.to_s == string
07. end
```

Métodos

► Métodos destrutivos

```
01. #método normal
02. a = "String velha"
03. a.sub("velha","nova") # Retorna "String nova"
04. puts a # String velha
05. #metodo destrutivo
06. a.sub!("velha","nova")
07. puts a # String nova
```

Exercícios

- ▶ Sabendo que a distância da Terra até a Lua é 380.000 Km, faça um método que, dado o número de viagens da Terra até a Lua (ida e volta), calcule a distância total percorrida.
- ▶ Escreva um programa que diga quantos anos tem uma pessoa que já viveu 9790000000 segundos



Respostas

► 1

```
01. DIST_TERRA_LUA = 380_000
02.
03. def viagemLua(n_vagens)
04.   2 * n_vagens * DIST_TERRA_LUA
05. end
06.
07. puts "2 viagens para a Lua: #{ viagemLua(2) }"
```

► 2

```
01. # 24 horas * 60 minutos * 60 segundos
02. SEGUNDOS_DIA = 24 * 60 * 60
03.
04. def segundos_to_ano(seg)
05.   seg/(SEGUNDOS_DIA * 365)
06. end
07.
08. segundos = 979000000
09. puts "Essa pessoa viveu #{segundos_to_ano(segundos)} anos."
```

Entrada de Dados

- ▶ Em nossos programas usaremos o método “gets” para entrada de dados via console
- ▶ Ele nos retorna um string com o texto digitado até que ENTER seja apertado
- ▶ Importante saber que o ‘\n’ (*line feed*) VAI estar presente na string recolhida



Entrada de Dados

▶ Como podemos usá-lo?

```
01. # exemplo simples
02. string = gets
03. # mas geralmente fazer assim é uma boa idéia
04. string = gets.chomp
05. #chomp removerá o último caractere da string (o \n)
```

▶ Outro exemplo

```
01. # depois de termos recolhido podemos converter para o tipo
02. # que precisamos
03. numero = gets.to_i
04. numero + 2
```

Conversões de dados

- ▶ **to_i** : converte para integer
- ▶ **to_s** : converte para string
- ▶ **to_f** : converte para float

Por enquanto não vamos nos preocupar com a consistência do que é digitado



Estruturas de Controle

► Como estruturas de controle temos:

- IF
- ELSIF
- UNLESS
- CASE



Estruturas de Controle

- ▶ Observemos o uso do if

```
01. #exemplo simples de if
02. var_bool = true
03.
04. if var_bool
05.   puts "Forever true"
06. else
07.   puts "Impossible!"
08. end
09.
10. var_num = 0
11. if var_num
12.   puts "0 é true?"
13. else
14.   puts "Impossible!"
15. end
16.
```

Somente FALSE e nil avaliam para falso em um teste

Estruturas de Controle

- ▶ Também temos o operador IF ternário, semelhante ao que existe em outras linguagens.

```
21. | if var_num ? "Verdade" : "Mentira"  
22. | # teste ? true : false
```

- ▶ Podemos optar por usar o if após a expressão de resposta, para deixar o código mais legível

```
01. | #uma outra construção do if no Ruby pode ser o if pós-fixado  
02.  
03. | puts "eh um numero" if 30.class == Fixnum  
04.
```

Estruturas de Controle

► Uso do ELSIF

```
01. =begin
02. elsif executa um else somente se a condição
03. dada for verdadeira.
04. Podemos encadeá-lo como quisermos.
05. =end
06.
07. nome = 'Lucas'
08. if nome == 'Bruno'
09.   puts 'eh o Bruno'
10. elsif nome == 'Lucas'
11.   puts 'eh o Lucas'
12. elsif nome == 'Jesus'
13.   puts 'aleluia!'
14. else
15.   puts 'entao quem eh?'
16. end
```

Estruturas de Controle

- ▶ Outra alternativa é a versão negada do IF, o UNLESS

```
07. unless true
08. puts "Equivalente a if false"
09. else
10. puts "Sempre essa saida"
11. end
12.
13. #da mesma forma
14.
15. puts "eh uma String" unless "String".class != String
```

Estruturas de Controle

- ▶ Para termos a mesma funcionalidade do SWITCH de outras linguagens, existe a construção CASE...WHEN

```
01. #Teste de case ... when
02.
03. puts 'Quantos anos voce tem?' ; STDOUT.flush
04. seletor = gets.to_i
05.
06. #note que o case nos retorna o valor encontrado
07.
08. escolha = case seletor
09.   when 0..10 then 'Kid'
10.   when 11..18 then 'Adolescente'
11.   when 19..65 then 'Adulto'
12.   when 66..100 then 'Idoso'
13.   else 'Tem certeza que a idade esta certa?'
14. end
15.
16. puts "Voce eh: #{escolha} "
```

Estruturas de Repetição

- ▶ Como estruturas de repetição temos:
 - WHILE
 - FOR
 - UNTIL
 - BEGIN
 - LOOP



Estruturas de Repetição – While

```
01. #exemplo de uso do while
02. i = 1
03.
04. while i < 6
05.   puts "Repetindo o laço... #{i}a. vez"
06.   i += 1
07. end
08.
09. =begin
10. Vai exibir:
11. Repetindo o laço... 1a. vez
12. Repetindo o laço... 2a. vez
13. Repetindo o laço... 3a. vez
14. Repetindo o laço... 4a. vez
15. Repetindo o laço... 5a. vez
16. =end
```

Estruturas de Repetição – For

```
01. #exemplo de uso do for
02.
03. for i in (1..6)
04. puts "Laco com for. #{i}a. iteração"
05. end
06.
07. =begin
08. Vai exibir:
09. Laco com for. 1a. iteração
10. Laco com for. 2a. iteração
11. Laco com for. 3a. iteração
12. Laco com for. 4a. iteração
13. Laco com for. 5a. iteração
14. =end
```

Estruturas de Repetição – For

```
01. #exemplo de uso do for com um array
02.
03. array = ["Bruno", "Lucas", "Ruby", "Paint", "Firefox"]
04. for i in array
05. puts i
06. end
07.
08. =begin
09. Vai exibir:
10. Bruno
11. Lucas
12. Ruby
13. Paint
14. Firefox
15. =end
```

Estruturas de Repetição – For

```
01. #exemplo de uso do for com um range de caracteres
02.
03. for c in ('j'..'p')
04. puts c
05. end
06.
07. =begin
08. Vai exibir:
09. j
10. k
11. l
12. m
13. n
14. o
15. p
16. =end
```

Estruturas de Repetição – Until

```
01. #exemplo de uso do until
02.
03. i = 0
04.
05. until i == 5 #faça até que...
06. puts i
07. i += 1
08. end
09.
10. =begin
11. Vai exibir:
12. 0
13. 1
14. 2
15. 3
16. 4
17. =end
```

Estruturas de Repetição – Begin

```
01. #exemplo de uso do begin (pode ser ser
02. #combinado com o while ou o until)
03.
04. #Corresponde ao Do...While de outras linguagens
05.
06. i = 0
07.
08. begin
09. puts i
10. i += 1
11. end while i < 5
12.
13. =begin
14. Vai exibir:
15. 0
16. 1
17. 2
18. 3
19. 4
20. =end
```

Estruturas de Repetição – Loop

```
01. #exemplo de uso do loop
02.
03. i = 0
04.
05. loop do
06.   break unless i < 5
07.   puts i
08.   i += 1
09. end
10.
11. =begin
12. Vai exibir:
13. 0
14. 1
15. 2
16. 3
17. 4
18. =end
```

Fixnum e Bignum

- ▶ Herdado da classe Integer, os dois representam números inteiros, com a diferença na magnitude do mesmo
- ▶ Fixnum trata de números até 31 bits e Bignum trata números maiores
- ▶ Ruby cuida da conversão entre eles



Arrays

- ▶ Arrays em Ruby são um pouco diferentes do que vocês podem estar acostumados e incluem vários métodos interessantes
- ▶ É preciso saber que em Ruby os arrays podem conter objetos de mais de uma classe ao mesmo tempo



Arrays

```
01. array = Array.new
02. #tb podemos declarar assim
03. array2 = []
04.
05. array.push(3)
06. array.push(4)
07. array.push("String") # é possivel ter tipos diferentes no mesmo array
08. array[2] = 30 #podemos acessar pelo índice
09.
10. puts array #[4,3,30]
```

Arrays

```
01. array = []
02. 10.times { |x| array.push(x) }
03. #alternativamente poderíamos fazer array<<x ou array[x] = x ou array += [x]
04.
05. array[10] == nil
06. #true, porque posições fora da faixa do array retornarão 'nil'
07.
08. var = array[-1]
09. puts var # imprime 9 porque acessamos o último elemento do array
10.
11. #e por fim, como podemos percorrer todos os elementos do array?
```

Arrays

```
01. #assim é o método 'tradicional' de fazê-lo
02. n = 0
03. while n < 10
04.   puts array[n]
05.   n += 1
06. end
07.
08. #mas em Ruby podemos usar
09. array.each { |n| puts n }
```

Arrays

- ▶ O método `each` é um exemplo de um iterator
- ▶ O essencial em Ruby é evitar usar os loops da maneira como estamos acostumados em outras linguagens
- ▶ A estrutura do próprio objeto deve nos permitir iterar sobre ele



Blocos de Código

- ▶ Uma das grandes *features* de Ruby são os blocos de código. Eles funcionam como as funções anônimas (*closures*) de outras linguagens
- ▶ Já usamos eles várias vezes



Blocos de Código

- ▶ Observe aqui que blocos de código são como funções que são definidas e usadas imediatamente.
- ▶ Note que se tornam muito poderosos combinados com outros aspectos de Ruby

```
01. #Como visto nos arrays
02. array = [3,5,6,7,3,1,2,3]
03. produtorio = array.inject do |x,y|
04.   x * y
05. end
06. #Iterando sobre Fixnums
07. 20.times do |num|
08.   puts "down" * num
09. end
```

Blocos de Código

- ▶ Lembrando dos métodos `each` e `times` que usamos anteriormente. Eles recebem um bloco de código e o executam sob algumas circunstâncias
- ▶ O core do Ruby está cheio deles, mas também podemos escrever os nossos



Exercícios

- ▶ Crie um método que receba um array como parâmetro e o preencha com strings inseridas pelo usuário até ser digitada a palavra ‘pare’. Use esse array para preencher dois arrays e imprima a intersecção dos dois



Respostas

```
01. def preencher_array(array)
02.     string = gets.chomp
03.     while string != 'pare'
04.         array << string
05.         string = gets.chomp
06.     end
07. end
08.
09. string = '' ; array = [] ; array2 = []
10. puts "Lendo primeiro array..."
11. preencher_array(array)
12. puts "\nLendo segundo array..."
13. preencher_array(array2)
14.
15. puts "\nInterseccão dos dois arrays:"
16. array.each {|str| puts str if array2.include?(str)}
```

Métodos para Arrays

- ▶ **sort** : [5, 1, 6, 2].sort => [1, 2, 5, 6]
- ▶ **include?** : ["um","dois"].include?("um") => true
- ▶ **all?** : [1, 5, 20].all? {|x| x>0} => true
[1, 5, 20].all? {|x| x>1} => false
- ▶ **any?** : [1, 5, 20].any? {|x| x>10} => true
[1, 5, 20].any? {|x| x>30} => false
- ▶ **length** : ['p','e','t'].length => 3
- ▶ **empty** : [].empty => true



Métodos para Strings

- ▶ **capitalize**: “ruby”.capitalize => “Ruby”
- ▶ **reverse**: “ruby”.reverse => “ybur”
- ▶ **upcase**: “ruby”.upcase => “RUBY”
- ▶ **downcase**: “RUBY”.downcase => “ruby”
- ▶ **length**: “ruby”.length => 4
- ▶ **swapcase**: “RuBy”.swapcase => “rUbY”
- ▶ **next**: “ruby”.next => “rubz”
- ▶ **empty?**: “ruby”.empty? => false
- ▶ **chop**: “ruby”.chop => rub
- ▶ **include?**: “ruby”.include?”ub” => true



Symbols

- ▶ Símbolos podem ser considerados como “strings mais leves”, mas elas são mais do que isto
- ▶ Diferentemente das strings, se usarmos um símbolo com o mesmo texto em mais de um contexto diferente, ele referenciará o mesmo dado na memória



Symbols

► Exemplo

```
01. |   a = "String" ; b = "String"
02. |   c = :symbol ; d = :symbol
03. |   a.object_id == b.object_id #false
04. |   c.object_id == d.object_id #true
```

Symbols

- ▶ Vários dados internos de Ruby são representados por símbolos
- ▶ Symbols são muito usados em Rails
- ▶ Para converter uma string para símbolo podemos usar o método `.to_sym`
- ▶ Ao estudarmos Hashes entenderemos um pouco mais sobre a importância de se usar símbolos



Hashes

- ▶ Podemos imaginar Hashes como arrays que podem ser indexados não apenas por inteiros, mas por qualquer objeto
- ▶ Importante notar: temos ‘keys’ (o que indexa) e ‘values’ (o que é indexado)
- ▶ Usar Hashes é uma prática comum em Rails



Hashes

▶ Por exemplo

```
01. hash = {} # ou Hash.new
02. hash['nome'] = 'Matz'
03. hash['linguagem'] = 'Ruby'
04. hash['pais'] = 'Japao'
05. hash.each do |key,value|
06.   puts "#{key} : #{value}"
07. end
08.
09. #Porem um método mais limpo de escrever isso é
10. hash = { 'nome' => 'Matz', 'linguagem' => 'Ruby', 'pais' => 'Japao' }
11.
12. #Use como um array
13. puts "#{hash['nome']} criou #{hash['linguagem']} no(a) #{hash['pais']}."
14. #Matz criou Ruby no(a) Japao.
```

Hashes

- ▶ É comum indexar hashes por strings, porém existe uma desvantagem nesse processo
- ▶ Imagine agora criar 30 hashes desse tipo. Note que alocaríamos 30 vezes cada uma das chaves
- ▶ O que nos importa são os dados, por isso não queremos gastar memória com as chaves



Hashes

- ▶ A solução é utilizar símbolos como chaves do Hash. Mesmo que tenhamos centenas de Hashes com as mesmas chaves, todas elas referenciarão os mesmos símbolos na memória.

```
01. #o que há de errado aqui?  
02. hash = {'nome' => 'Matz', 'linguagem' => 'Ruby', 'pais' => 'Japao'}  
03. hash2 = {'nome' => 'Guido', 'linguagem' => 'Python', 'pais' => 'Paises Baixos'}  
04. #cada novo hash, aloca novamente as strings que são usadas como keys  
05.  
06. #solução  
07. hash = { :nome => 'Matz', :linguagem => 'Ruby', :pais => 'Japao'}
```

Métodos para Hashes

- ▶ **merge:** hash.merge(outro_hash) => retorna um hash com a união dos dois hashes passados
- ▶ **merge!:** hash.merge!(hash) => igual a ‘merge’, mas modifica o objeto original
- ▶ **size:** hash.size => retorna o número de elementos do hash
- ▶ **has_key?:** hash.has_key?(:Ruby) => true se possuir a chave passada
- ▶ **has_value?:** hash.has_value?(:Ruby) => true se a chave possuir valor

Criando nossas próprias Classes

- ▶ Um objeto é uma ‘entidade’ que serve de container para informações e também controla o acesso a elas.
- ▶ O objeto possui uma série de atributos que, essencialmente, não são mais do que variáveis que pertencem a este objeto.
- ▶ O mesmo terá associado a ele várias funções (métodos), que fornecem uma interface para utilizá-los.



Criando nossas próprias Classes

- ▶ Em Ruby a criação de novas classes é muito simples

```
01. class Cachorro
02.   def initialize(nome,raca)
03.     @nome = nome
04.     @raca = raca
05.   end
06. end
```

Criando nossas próprias Classes

- ▶ Notem o método “initialize”. Ele é o construtor das instâncias da nossa classe. Nós especificamos que ele recebe dois parâmetros, nome e raça e os atribui a duas variáveis de classe.
- ▶ As variáveis de classe são declaradas com uma ‘@’ na frente



Criando nossas próprias Classes

- ▶ Aqui uma implementação da classe Cachorro

```
01. class Cachorro
02.   def initialize(nome, raca)
03.     @nome = nome
04.     @raca = raca
05.   end
06.
07.   def latir
08.     puts "#{@nome}: AU AU AU!"
09.   end
10.
11.   def perseguir_gato
12.     puts "#{@nome} está perseguindo um gato!"
13.   end
14.
15.   def comer(oque)
16.     puts "#{@nome}: Au Au barf Au! (Que #{oque} delicioso)"
17.   end
18. end
```

Criando nossas próprias Classes

- ▶ E podemos usá-lo da seguinte maneira:

```
20. # .new aloca espaço para a nossa instância
21. # e chama o nosso initialize
22. cachorro1 = Cachorro.new('Kaflubes', 'Belga Holandes')
23. cachorro2 = Cachorro.new('Rex', 'Vira-lata')
24.
25. #usando os métodos do nosso cachorro
26. cachorro1.latir
27. cachorro2.comer('Purê')
28. cachorro2.perseguir_gato
```

Exercícios

1. Crie uma classe SuperHero, que possui como dados o seu nome, sua identidade secreta e um array com seus super-poderes. A classe deve incluir um método chamado combater_crime e outros a gosto
2. Use os métodos da classe criada para ver se estão funcionando



Resposta

```
01. class SuperHero
02.   def initialize nome,id_secreta, poderes
03.     @nome = nome
04.     @id_secreta = id_secreta
05.     @poderes = poderes
06.   end
07.
08.   def combateCrime
09.     "#{@nome} combatendo o crime."
10.   end
11. end
```

Criando nossas próprias Classes

- ▶ Pegando a classe Cachorro que fizemos anteriormente. E se quisermos acessar os dados dos nossos objetos?
- ▶ Métodos setter e getter?



Criando nossas próprias classes

```
01. class Cachorro
02. ...
03. def nome
04.   @nome
05. end
06.
07. def nome=(n)
08.   @nome = n
09. end
10.
11. def raca
12.   @raca
13. end
14.
15. def raca=(r)
16.   @raca = r
17. end
```

Criando nossas próprias Classes

- ▶ Usando:

```
01. c = Cachorro.new("Kaflubes", "Belga Holandes")
02. puts "#{c.nome} é um #{c.raca}"
03. c.nome = "Fido"
04. c.raca = "Labrador"
05. puts "#{c.nome} é um #{c.raca}"
```

- ▶ Mas Ruby pode nos fornecer setters e getters automaticamente...

Criando nossas próprias Classes

- ▶ Observe:

```
01. class Cachorro
02.     attr_accessor :nome, :raca
03.
04.     def initialize(nome,raca)
05.         @nome = nome
06.         @raca = raca
07.     end
08.     ...
09. end
```

- ▶ Com esta construção teremos efetivamente o mesmo resultado

Criando nossas próprias Classes

- ▶ Podemos usar:

- attr_accessor: cria um getter e setter
- attr_reader: cria somente o getter
- attr_writer: cria somente o setter



Criando nossas próprias Classes

- ▶ Vamos agora tentar “imprimir” o nosso cachorro

```
24. | cachorro = Cachorro.new("Kxorro", "Vira-lata")
25. | puts cachorro # tentem fazer isso aí
```

- ▶ O que aconteceu? O método ‘puts’ converte o objeto para String. Como o nosso Cachorro é um objeto e herda de Object, ele chama o método `to_s` que está definido na classe Object

Criando nossas próprias Classes

- ▶ Então, podemos facilmente fazer uma impressão mais bonita do nosso cachorro, redefinindo o método `to_s`:

```
01. class Cachorro
02. ...
03.   def to_s
04.     "Cachorro:\n * Nome: #{@nome}\n * Raça: #{@raca}"
05.   end
06. end
```

Incluindo classes e Bibliotecas

- ▶ Depois que já tivermos a nossa classe pronta, podemos salvá-la como um arquivo .rb
- ▶ Usando o comando ‘require arquivo’ (sem a extensão) podemos incluir um arquivo e usar suas classes e funções.
- ▶ O comando require procura arquivos no diretório em que o programa está sendo executado e no diretório padrão do Ruby

Herança

- ▶ Um aspecto importantíssimo em Ruby (e em qualquer linguagem orientada a objetos) é poder criar classes a partir das que nós já temos. No jargão da OO essa técnica é denominada herança.
- ▶ Vamos agora criar a classe SpiderMan a partir do SuperHero que já está pronta



Herança

- ▶ A herança é dada pelo operador “<”
- ▶ Notem que a classe “filha” recebe todos os métodos da classe “pai”

```
01. class Spiderman < SuperHero  
02.  
03.   def initialize nome,id_secreta,poderes,cor_da_roupa  
04.     super(nome,id_secreta,poderes)  
05.     @cor = cor_da_roupa  
06.   end  
07.  
08.   def tirar_fotos  
09.     puts "*Spidey tirando fotos*"  
10.   end  
11. end
```

Herança

- ▶ Podemos redefinir métodos da classe pai na classe filha.
- ▶ O método super, quando usado dentro de um método, chama o método correspondente da classe pai.
- ▶ No exemplo anterior chamamos o método initialize da classe SuperHero

