Linguagens de Programação – 2015/1

Ruby

Diego Rodrigues
Leonardo Rodrigues

Introdução

- Criada em 1995 pelo japonês *Yuri Matsumoto*
- Uniu partes das suas linguagens favoritas: *Perl*, *Smalltalk*, *Eiffel*, *Ada* e *Lisp*
- Equilíbrio entre programação *funcional* e *imperativa*
- Uma linguagem multi-plataforma
- Em 2006 atingiu aceitação massiva
- Open Source

Características

- Linguagem alto nível
- Interpretada
- Orientada à objetos
- Fácil de usar
- Trabalha com herança, classes, métodos, polimorfismo e escalonamento
- Tipagem dinâmica

Instalação

- Compatível com os sistemas *Linux/Unix*, *OS X* e *Windows*
- Download e instruções disponíveis em:
  https://www.ruby-lang.org/en/documentation/installation/
- apt package manager (Debian or Ubuntu)

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

Execução de um programa

- Executar um código em Ruby é bem simples!

```ruby
1  puts "Hello World!"
```

```
leonardo@leonardo-linux: ~/Desktop/programa_ruby
leonardo@leonardo-linux:~/Desktop/programa_ruby$ ruby code.rb
Hello World!
leonardo@leonardo-linux:~/Desktop/programa_ruby$
```

Sintaxe

- A Sintaxe do Ruby é simples e exata. Sem necessidades de caracteres de término de uma instrução
  - Possui boa legibilidade
  - Fácil escrita

```ruby
x = 10
y = 9.2
name = "Leonardo"

puts "#{name}, #{x}, #{y}"

print "Alguma string aqui!"
```

```
leonardo@leonardo-linux: ~/Desktop/programa_ruby
leonardo@leonardo-linux:~/Desktop/programa_ruby$ ruby code.rb
leonardo, 10, 9.2
Alguma string aqui!leonardo@leonardo-linux:~/Desktop/programa_ruby$
```

Sintaxe – Variáveis

- Variáveis de instância

  @cliente

- Variáveis de Classe

  @@cliente

- Variáveis Globais

  $cliente

- Variáveis Locais

  cliente

Sintaxe – Variáveis

- Exemplificando:

```
1 opcao = "sim"    #locais
2 $versao = 1.5    #globais
3 @idade = 19      #instancia
4 @@cont = 28      #classes
```

Sintaxe – Comentários

- Em Ruby há duas formas de se fazer comentários:

```
1  # your comment here
2
3  =begin
4       your comment here
5       and here
6         .
7         .
8         .
9  =end
```

Sintaxe - Métodos

- A criação de métodos também é bem simples

```ruby
def meu_metodo (n)
    x = 10**n
    puts "#{x}"
end

meu_metodo(3)
```

Sintaxe - Conversão

- Podemos fazer uma conversão explícita das variáveis

```
1  x = 2.5
2  y = x.to_i
3
4  z = 5
5  string = z.to_s
6
7  puts "#{string}"
8
9
```

Sintaxe - Operadores

- Operadores que coincidem com o padrão de outras linguagens

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo
**	Exponencia

Sintaxe - Operadores

- Assim como os operadores aritméticos, os operadores relacionais também são métodos e seguem a mesma lógica

==	Igual
!=	Diferente
>	Maior
<	Menor
>=	Maior ou igual
<=	Menor ou igual

Sintaxe - Operadores

- Operadores Condicionais

and / &&	e
or / ||	ou
not	negação

Sintaxe - Operadores

- Operadores Bitwise. Fazem operação bit-a-bit

&	e
|	ou
^	ou exclusivo
!	negação
~	complemento de 1
<<	deslocamento à esquerda
>>	deslocamento à direita

Sintaxe - Operadores

- Operadores Bitwise. Fazem operação bit-a-bit

```ruby
a = 60
b = 13

result = a&b

puts "#{result}"
```

```
diegopr@ubuntu: ~/Desktop
diegopr@ubuntu:~/Desktop$ ruby code.rb
12
diegopr@ubuntu:~/Desktop$
```

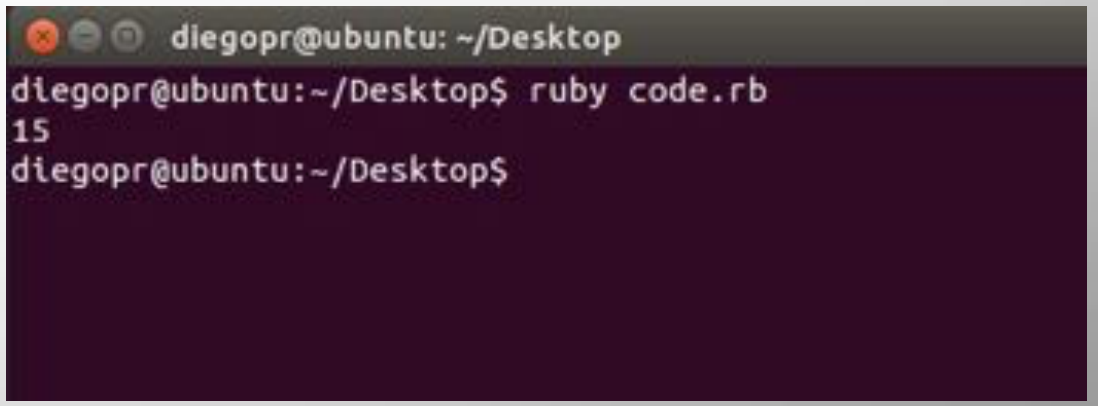
Sintaxe - Operadores

- Operador Ternário

```ruby
x = 5

(x == 4) ? (y = 10) : (y = 15)

puts "#{y}"
```



Sintaxe – Palavras Reservadas

- 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

Tipagem

- Dinâmica
- Implicitamente tipada
- Ruby detecta o tipo de cada variável em tempo de execução

Tipagem – Dinâmica Forte

- Não necessita de declarar variáveis. Sendo assim o ruby identifica o tipo dos caracteres. É considerado tipagem forte pois não aceita as misturas de tipos.

- O ruby não aceita misturas como:

```
a = “a”
b = 1
c = a+b
```

Tipos Básicos

- Blocos de Código
- Números
- Booleanos
- Strings
- Constantes
- Ranges
- Array
- Símbolos
- Expressões Regulares
- Procs

Estruturas de Controle

- Ruby não se importa com espaços e/ou linhas em branco então a indentação da instrução *puts* não é necessária.

Estruturas de Controle

- As vezes você quer usar o fluxo de controle para verificar se algo é falso, ao invés de verdadeiro. Você poderia reverter seu if/else, mas Ruby tem uma alternativa melhor: ele vai deixar que você use uma instrução *unless*.

Estruturas de Controle

- O case

script.rb

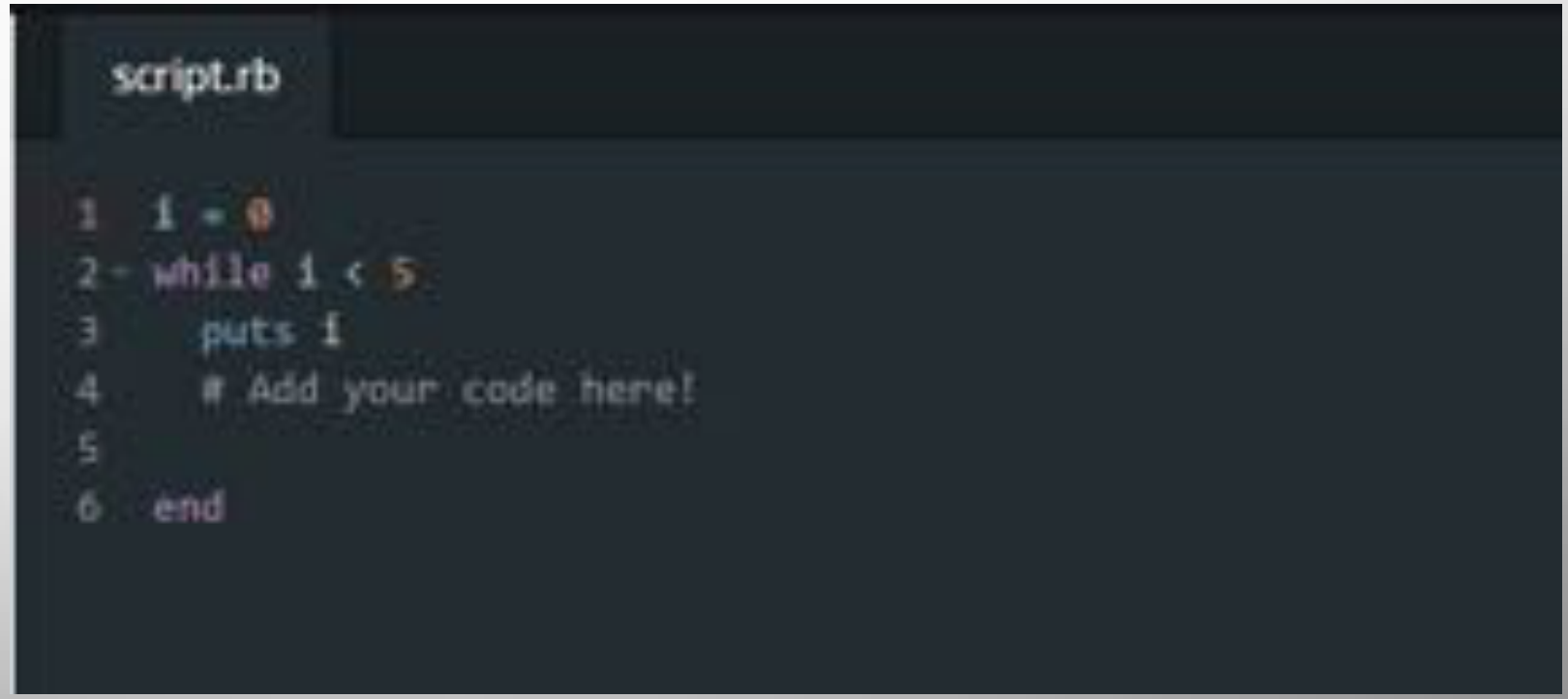
```ruby
case i
    when 0..5
        puts "Esta entre 0 e 5"
    when 6..10
        puts "Esta entre 6 e 10"
    else
        puts i.to_s
end
```

Loops & Iterações

- While

Loops & Iterações

- While



Perigo! Loop infinito!

Loops & Iterações

- Until

Loops & Iterações

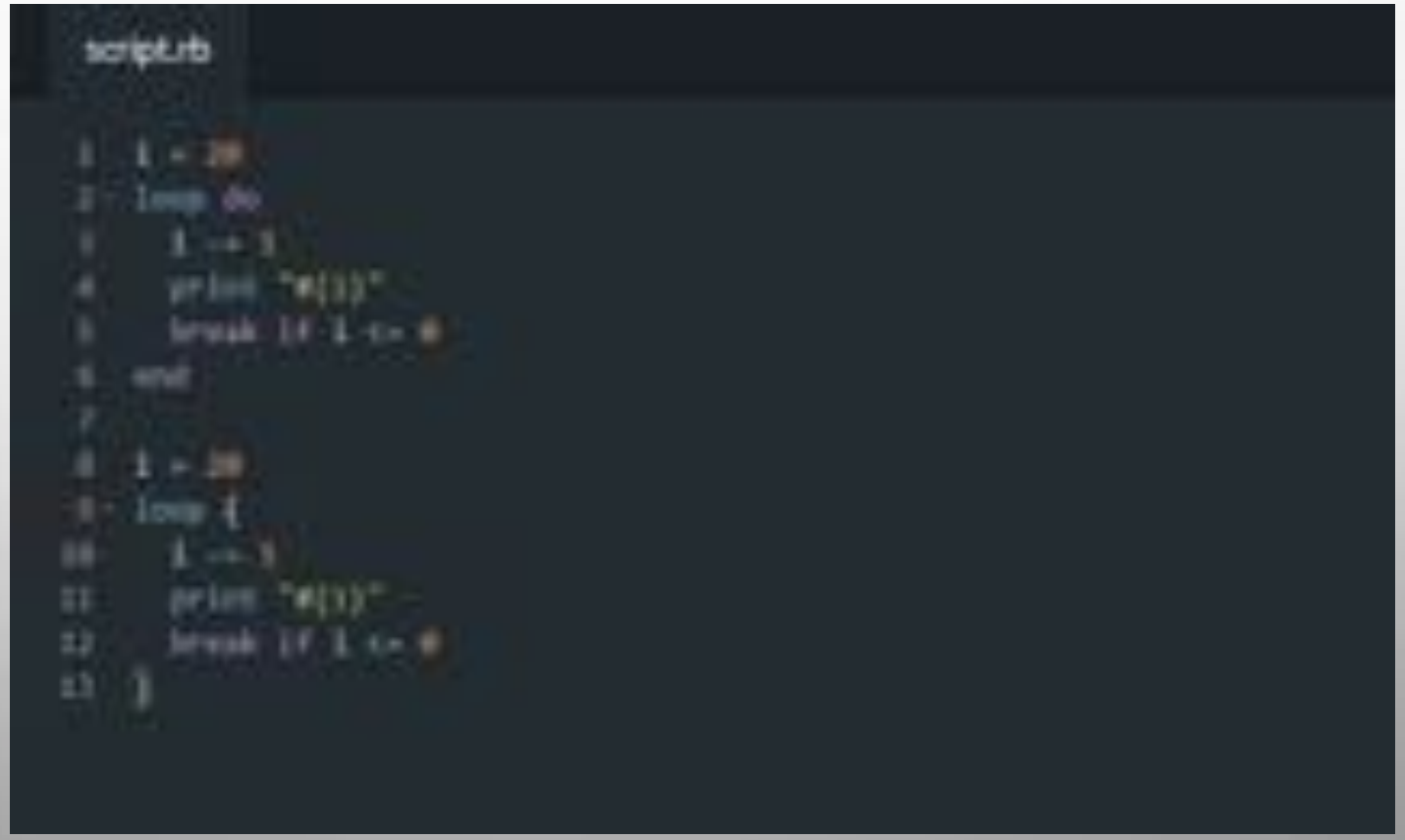
- For

script.rb

```ruby
for num in 1...10
    puts num
end
```

Loops & Iterações

- Loop



Como quase tudo em Ruby há mais de uma maneira de fazer as coisas. Neste caso temos o método *loop*, bastante eficaz porém perigoso por não ter condição de parada. Logo, o uso do *break* se faz necessário.

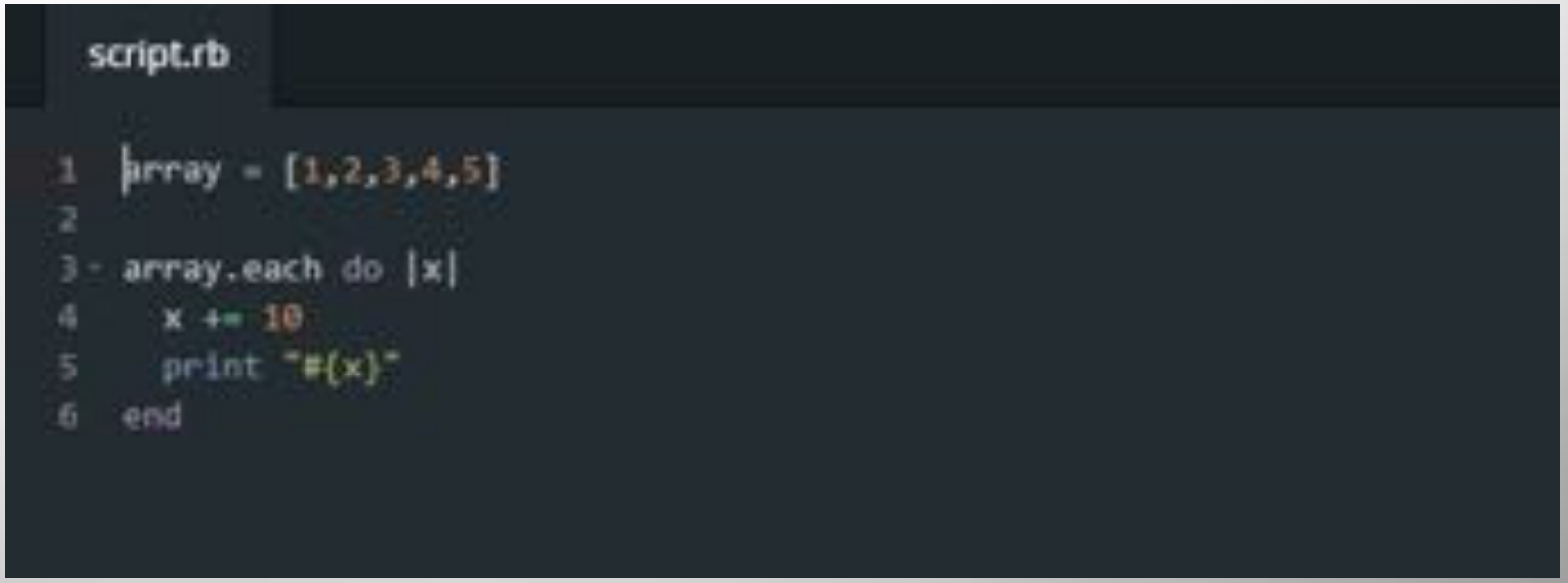
Loops & Iterações

- Next!

A palavra reservada *next* pode ser usada para pular certas etapas dentro do loop. No caso acima, se não quisermos printar números ímpares, pulamos a iteração.

Loops & Iterações

- O iterador .each



Um dos mais utilizados em Ruby, o poderoso método iterador .*each* pode manipular cada elemento de um objeto individualmente.

Loops & Iterações

- O iterador .times

script.rb

```
1 - 10.times do
2       print "Jimi Newdo"
3   end
```

O método iterador *.times* pode-se comparar como um super compacto iterador *for*. Executa uma tarefa para cada elemento de um objeto em um determinado número de vezes.

Estrutura Array

- Em Ruby, a estrutura pré-definida Array é utilizada para a a definição de vetores
- Vetores alocados dinamicamente com possobilidade de se ter vários tipos simultaneamente

script.rb

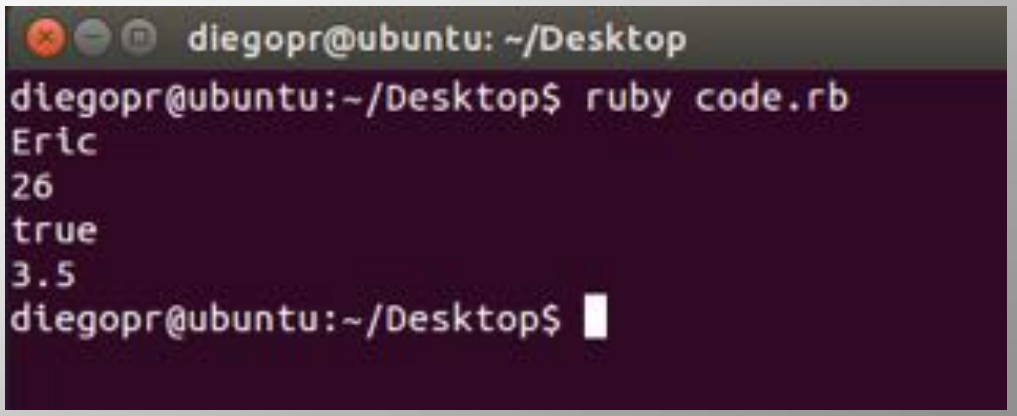
```
1  array_A = [1, 2, 3]
2
3  array_B = [1, 2.5, "String"]
4
5  for i in 0...3
6      puts"array_A[#{i}] = #{array_A[i]}"
7  end
8
9  for i in 0...3
10     puts"array_B[#{i}] = #{array_B[i]}"
11 end
```

diegopr@ubuntu: ~/Desktop

```
diegopr@ubuntu:~/Desktop$ ruby code.rb
array_A[0] = 1
array_A[1] = 2
array_A[2] = 3
array_B[0] = 1
array_B[1] = 2.5
array_B[2] = String
diegopr@ubuntu:~/Desktop$
```

Estrutura Hash

- A estrutura Hash funciona como um dicionário onde se é possível armazenar diferentes tipos de valores, sendo cada um deles associados à uma chave em questão



Orientação à Objetos

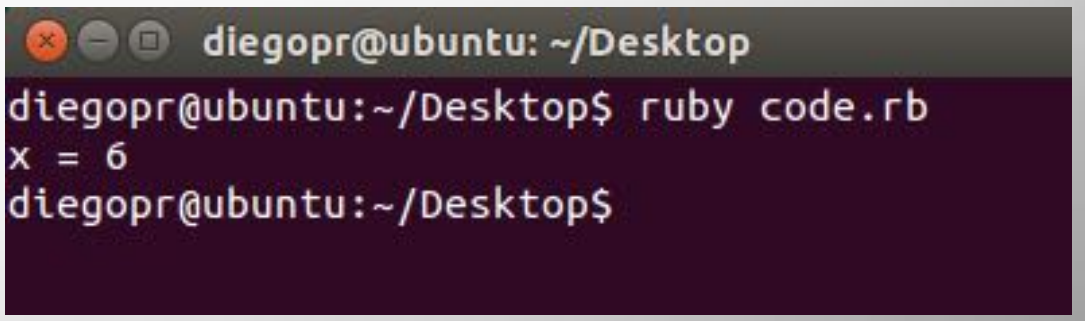
- Quase tudo em Ruby é um objeto!

script.rb

```ruby
x = "string".length

puts "x = #{x}"

```



Orientação à Objetos - Uso

- Exemplo de definição e uso de classes, métodos e atributos

```
diegopr@ubuntu: ~/Desktop
diegopr@ubuntu:~/Desktop$ ruby code.rb
I'm Ruby and I was created by Yukihiro Matsumoto!
I'm Python and I was created by Guido van Rossum!
I'm JavaScript and I was created by Brendan Eich!
diegopr@ubuntu:~/Desktop$
```

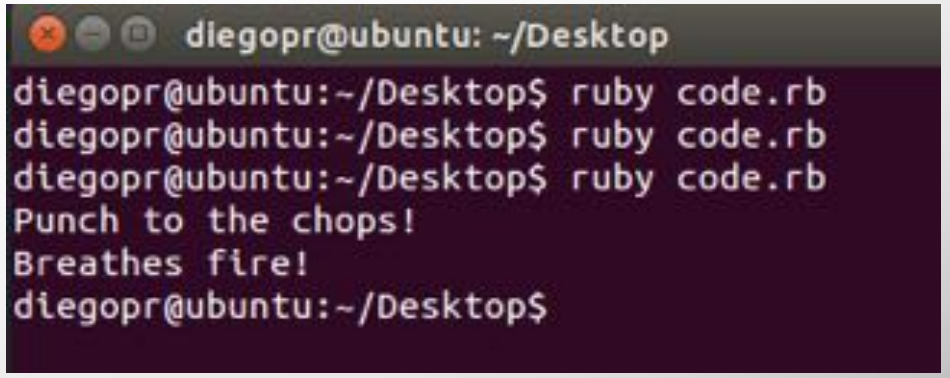
Orientação à Objetos - Uso

- Em Ruby, também se faz possível o uso de métodos e atributos como sendo *public, private* ou *protected*
- Por omissão, os métodos são definidos como sendo *public* e os atributos como sendo *private*
- Relembrando...

@cliente - Variáveis de instância
@@cliente - Variáveis de Classe
$cliente - Variáveis Globais
cliente - Variáveis Locais

Orientação à Objetos - Sobrescrita

- A utilização do mecanismo de Sobrecarga em Ruby se faz possivel da seguinte maneira:



Orientação à Objetos - Herança

- A utilização do mecanismo de Herança em Ruby se faz possivel da seguinte maneira:

script.rb

```ruby
class ApplicationError
  def display_error
    puts "Error! Error!"
  end
end

class SuperBadError < ApplicationError
end

err = SuperBadError.new
err.display_error
```

diegopr@ubuntu: ~/Desktop

```
diegopr@ubuntu:~/Desktop$ ruby code.rb
Error! Error!
diegopr@ubuntu:~/Desktop$
```

Orientação à Objetos – Herança Múltipla

- Ruby não suporta o uso do mecanismo de Herança Múltipla, porém oferece o uso de *Mixins* para emular o mesmo

```
class Pessoa < Mamifero # Herança de Mamifero
        include Humano # Emulando herança múltipla
end
```

Modules

- Além das classes normais, Ruby possui os "Modules", que funcionam como se fosse classes de classes

Modules - Uso

```
diegopr@ubuntu:~/Desktop$ ruby code.rb
I jumped forward 3 feet!
I jumped forward 3 feet!
diegopr@ubuntu:~/Desktop$
```

Polimorfismo

- Em Ruby os quatro tipos de Polimorfismo se fazem presente,
  - Polimorfismo de Coerção
  - Polimorfismo de Sobrecarga
  - Polimorfismo Paramétrico
  - Polimorfirsmo por inclusão

Exceções

- Uma exceção é um tipo especial de objeto, uma instância da classe **Exception** ou uma descendente dessa classe que representa um tipo de condição incomum; ela indica que algo de errado aconteceu. Quando isso ocorre, uma exceção é lançada. Por padrão, os programas Ruby terminam quando uma exceção ocorre, mas é possível declarar tratadores de exceção (*handlers*).

- Lançar uma exceção significa parar a execução normal do programa e transferir o fluxo de controle para o código tratador da exceção onde você pode ou lidar com o problema que foi encontrado ou sair completamente do programa. O que irá acontecer – lidar com o problema ou abortar o programa – depende do que você forneceu um comando de resgate (**rescue**, que é uma parte fundamental da linguagem Ruby). Se você não forneceu este comando, o programa termina; se você forneceu, o fluxo de controle é passado ao comando **rescue**.

Exceções

- O Ruby tem algumas classes pré-definidas – Exceptions e suas filhas – que lhe ajudam a lidar com erros que ocorrem em seu programa. A figura ao lado mostra a hierarquia das exceções em Ruby.

Exceções – Lançando uma exceção

- O seguinte método lança uma exceção sempre que é chamado. Sua segunda mensagem nunca será impressa.

```ruby
# p043raise.rb
def levanta_excecao
  puts 'Estou antes do raise.'
  raise 'Um erro ocorreu'
  puts 'Estou depois do raise'
end
levanta_excecao
```

```
>ruby p043raise.rb
Estou antes do raise.
p043raise.rb:4:in `levanta_excecao': Um erro ocorreu (RuntimeError)
    from p043raise.rb:7
>Exit code: 1
```

Exceções – Lançando um exceção

- O método **raise** é do módulo Kernel. Por padrão, **raise** cria uma exceção da classe **RuntimeError**. Para levantar a exceção de uma classe específica, você pode passar o nome da classe como argumento para **raise**.

```ruby
# p044inverse.rb
def inverte(x)
  raise ArgumentError, 'O argumento não é numérico' unless x.is_a? Numeric
  1.0 / x
end
puts inverte(2)
puts inverte('não é um número')
```

```
>ruby p044inverse.rb
0.5
p044inverse.rb:3:in `inverte': O argumento não é numérico (ArgumentError)
    from p044inverse.rb:7
>Exit code: 1
```

Exceções – Lançando uma exceção

- Você ainda pode definir sua própria subclasse de exceção:

```
1   class NotInvertibleError < StandardError
2   end
```

Exceções – Tratando uma exceção

- Para fazer o tratamento da exceção, nós colocamos o código que pode lançar uma exceção dentro de um bloco **begin-end** e usamos uma ou mais cláusulas **rescue** para contar ao Ruby os tipos de exceção que queremos tratar.

```
1   # p045handexcp.rb
2   def levanta_e_resgata
3     begin
4       puts 'Estou antes do raise.'
5       raise 'Um erro ocorreu'
6       puts 'Estou depois do raise'
7     rescue
8       puts 'Fui resgatado.'
9     end
10    puts 'Estou depois do bloco begin.'
11  end
12  levanta_e_resgata
```

```
1   >ruby p045handexcp.rb
2   Estou antes do raise.
3   Fui resgatado.
4   Estou depois do bloco begin.
5   >Exit code: 0
```

Exceções – Lançando uma exceção

- Você pode empilhar cláusulas de **rescue** em um bloco begin/rescue. Exceções não tratadas por uma cláusula rescue serão passadas para a próxima:

```
1  begin
2    # -
3  rescue UmTipoDeExcecao
4    # -
5  rescue OutroTipoDeExcecao
6    # -
7  else
8    # -
9  end
```

Exceções – Lançando uma exceção

- 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, como mostrado no programa.

```
1   # p046excpvar.rb
2   begin
3     raise 'Uma exceção para teste.'
4   rescue Exception => e
5     puts e.message
6     puts e.backtrace.inspect
7   end
```

```
1   >ruby p046excpvar.rb
2   Uma exceção para teste.
3   ["p046excpvar.rb:3"]
4   >Exit code: 0
```

Avaliação da Linguagem

Concorrência	Sim (Thread, semáforos)
Confiabilidade	Média
Custo	Baixo (linguagem free)
Eficiência	Baixa (interpretada)
Exceções	Sim
Facilidade de aprendizado	Boa
Gerência de Memória	Garbage Colector
Integração	Java, C#, PhP
Legibilidade	Boa
Método de Projeto	Interpretada, multiparadigma (OO, imperativa, refle)
Passagem de parâmetros	Cópia
Portabilidade	Alta (Linux, Mac, Windows)
Redigibilidade	Excelente
Reusabilidade	Alta
Verificação de tipos	Tipagem dinâmica e forte