Seminário Ruby

- Alunos
  - Hudson Martins da Silva
  - Laércio
  - Pedro Hoppe

1. Breve Histórico:

- Desenvolvida no Japão em 1995, por *Yukihiro "Matz" Matsumoto.*
  - Linguagem de script.
  - Mais poderosa do que Perl, e mais orientada a objetos do que Python.
  - Escrita em C
- Uniu partes das suas linguagens favoritas:
  - Perl, Smalltalk, Eiffel, Ada e Lisp.
- Equilibra a programação funcional com a programação imperativa.

Ruby
A Programmer's Best Friend

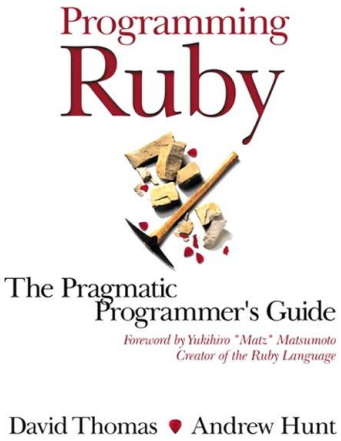
2. Ideais do Criador

- "*Tento tornar o Ruby natural, não simples*"
- "*O Ruby é simples na aparência, mas muito complexo no interior, tal como o corpo humano.*"
- Inicialmente foram propostos dois nomes: "Coral" e "Ruby"
- Matsumoto escolheu Ruby por ser a pedra zodiacal de um de seus colegas.

Totalmente livre. Não somente livre de custos, mas também livre para utilizar, copiar, modificar e distribuir.

3. Crescimento

* Lista de discussão em inglês chamada *Ruby-Talk*
  - O primeiro livro em inglês, *Programming Ruby*, liberado gratuitamente para o público, ajudou no processo de adoção de Ruby por falantes do inglês.
  - Por volta de 2005, o interesse pela linguagem Ruby subiu em conjunto com o Ruby on Rails, um framework de aplicações web popular escrito em Ruby.
  - Eleita Linguagem de Programação do Ano em 2006
  - RubyForge: repositório de projetos



5. Instalando a Linguagem

- Ruby é compatível com os sistemas Windows, Linux e Mac OS.
- Os códigos binários podem ser obtidos no site: https://www.ruby-lang.org/pt/downloads/
- No caso das versões Ubuntu/Debian do Linux e seus derivados, digite no terminal:

```
$ sudo apt-get install ruby irb rdoc
```

4. Principais Características

- Estrutura Geral:
  - Linguagem interpretada;
  - Orientada à objetos (“tudo” é objeto);
  - Portável;
  - Trabalha com herança, classes, métodos, polimorfismo e escalonamento;
  - Sintaxe relativamente simples e de fácil compreensão
  - Tipagem dinâmica mas forte
  - A sintaxe é enxuta, quase não havendo necessidade de colchetes e outros caracteres.
  - Linguagem de propósito geral

4. Principais Características

- Ruby possui, assim como Python, um modo interativo onde é possível interpretar linhas de códigos instantaneamente:

```
ncd@ncd-virtual-machine ~/Área de Trabalho $ irb
irb(main):001:0> def soma( a, b )
irb(main):002:1> a + b
irb(main):003:1> end
=> nil
irb(main):004:0> soma( 1, 3 )
=> 4
irb(main):005:0>
```

5. Sintaxe

- Sintaxe limpa, simples e exata.
- Leitura natural e fácil escrita.
- "*O Ruby é simples na aparência, mas muito complexo no interior, tal como o corpo humano.*"
- Linguagem de Máquina + Linguagem Natural
- Exemplo:

```
irb(main):001:0> puts "Olá"
Olá
=> nil
irb(main):002:0> [ 12, 47, 35 ].max
=> 47
```

5. Sintaxe – Comentários

- Para inserir cométarios basta iniciar a linha com '#'.
- Exemplo

```
# Programa para achar o fatorial de um número
# Nome do Arquivo: Fatorial.rb

def fact( n )
        if n == 0
                1
        else
                n * fact( n - 1 )
        end
end
```

5. Sintaxe – Palavras Reservadas

- As palavras reservadas da linguagem Ruby são:

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

5. Sintaxe – Variáveis

- Por a tripagem ser dinâmica, não precisamos declarar o tipo de variável:

```ruby
# Teste.rb
#Programa demonstrativo

a = 5
puts a
a = "Ola Mundo!\n"
puts a
a = 3.5
puts a
```

```
ncd@ncd-virtual-machine ~/Area de Trabalho $ ruby Teste.rb
5
Ola Mundo!
3.5
ncd@ncd-virtual-machine ~/Área de Trabalho $
```

5. Sintaxe – Variáveis

- Podemos fazer uma conversão explícita das variáveis usando os métodos to_i (para int), to_f (para float) e to_s (para string)

```
irb(main):001:0> "Vitor".reverse
=> "rotiV"
irb(main):002:0> 40.reverse
NoMethodError: undefined method `reverse' for 40:Fixnum
        from (irb):2
        from /usr/bin/irb:12:in `<main>'
irb(main):003:0> 40.to_s.reverse
=> "04"
irb(main):004:0>
```

5. Sintaxe – Strings

- As cadeias de caracteres (ou “strings”) são representadas por aspas duplas ou simples:

```
irb(main):001:0> "abc"
=> "abc"
irb(main):002:0> 'abc'
=> "abc"
irb(main):003:0>
```

- Podemos realizar operações sobre “strings”:

```
irb(main):003:0> "jo" + "ão"
=> "joão"
irb(main):004:0> "jo" * 2
=> "jojo"
irb(main):005:0> flor = "Rosa"
=> "Rosa"
irb(main):006:0> flor[ 0 ]
=> "R"
irb(main):007:0> flor[ 0, 2 ]
=> "Ro"
irb(main):008:0> flor[0..3]
=> "Rosa"
irb(main):009:0> "Rosa" == "Rosa"
=> true
irb(main):010:0> "Rosa" == "Carro"
=> false
```

5. Sintaxe – Operadores

- Ruby suporta um rico conjunto de operadores. A maioria dos operadores são na verdade chamadas de métodos. Por exemplo, “a + b” é interpretado como “a.+(b)”, onde o método + é invocado pelo objeto ‘a’ e tem como parâmetro de entrada ‘b’.

5. Sintaxe – Operadores

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20

5. Sintaxe – Operadores

Ruby Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
<=>	Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second.	(a <=> b) returns -1.
===	Used to test equality within a when clause of a *case* statement.	(1...10) === 5 returns true.
.eql?	True if the receiver and argument have both the same type and equal values.	1 == 1.0 returns true, but 1**.eql?**(1.0) is false.
equal?	True if the receiver and argument have the same object id.	if aObj is duplicate of bObj then aObj == bObj is true, a**.equal?**bObj is false but a**.equal?**aObj is true.

5. Sintaxe – Operadores

Ruby Assignment Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assigne value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a

5. Sintaxe – Operadores

Ruby Parallel Assignment:

Ruby also supports the parallel assignment of variables. This enables multiple variables to be initialized with a single line of Ruby code. For example:

```
a = 10
b = 20
c = 30
```

may be more quickly declared using parallel assignment:

```
a, b, c = 10, 20, 30
```

Parallel assignment is also useful for swapping the values held in two variables:

```
a, b = b, c
```

5. Sintaxe – Operadores

Ruby Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Ruby language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(a | b) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~a) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15, which is 0000 1111

5. Sintaxe – Operadores

Ruby Logical Operators:

There are following logical operators supported by Ruby language

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true, then the condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a or b) is true.
&&	Called Logical AND operator. If both the operands are non zero, then the condition becomes true.	(a && b) is true.
||	Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true.	(a || b) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(a && b) is false.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	not(a && b) is false.

Ruby Ternary operator:

There is one more operator called Ternary Operator. This first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation. The conditional operator has this syntax:

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

5. Sintaxe – Operadores

Ruby Range operators:

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value and a range of values in between.

In Ruby, these sequences are created using the ".." and "..." range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

Operator	Description	Example
..	Creates a range from start point to end point inclusive	1..10 Creates a range from 1 to 10 inclusive
...	Creates a range from start point to end point exclusive	1...10 Creates a range from 1 to 9

5. Sintaxe – Operadores

Ruby Operators Precedence

The following table lists all operators from highest precedence to lowest.

Method	Operator	Description
Yes	::	Constant resolution operator
Yes	[ ] [ ]=	Element reference, element set
Yes	**	Exponentiation (raise to the power)
Yes	! ~ + -	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
Yes	* / %	Multiply, divide, and modulo
Yes	+ -	Addition and subtraction
Yes	>> <<	Right and left bitwise shift
Yes	&	Bitwise 'AND'
Yes	^ |	Bitwise exclusive `OR' and regular `OR'
Yes	<= < > >=	Comparison operators
Yes	<=> == === != =~ !~	Equality and pattern match operators (!= and !~ may not be defined as methods)
	&&	Logical 'AND'
	||	Logical 'OR'
	.. ...	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= { /= -= += |= &= >>= <<= *= &&= ||= **=	Assignment
	defined?	Check if specified symbol defined
	not	Logical negation
	or and	Logical composition

NOTE: Operators with a *Yes* in the method column are actually methods, and as such may be overridden.

6. Estruturas de Controle – if ... else

- Semelhante ao if...else do C/C++ e Java

```
if (teste lógico)
        comandos
elsif
        comandos
   else
        comandos
   end
```

```
irb(main):027:0> def compara( a )
irb(main):028:1> if a == 0
irb(main):029:2> puts "A eh igual a zero"
irb(main):030:2> elsif a == 1
irb(main):031:2> puts "a eh igual a um"
irb(main):032:2> else
irb(main):033:2* "a eh diferente de 0 ou 1"
irb(main):034:2> end
irb(main):035:1> end
=> nil
irb(main):036:0> compara( 2 )
=> "a eh diferente de 0 ou 1"
irb(main):037:0> compara( 1 )
a eh igual a um
=> nil
```

6. Estruturas de Controle – Case

- Semelhante ao Switch-Case do C/C++ e Java

```
irb(main):001:0> i = 8
=> 8
irb(main):002:0> case i
irb(main):003:1> when 1, 2..5
irb(main):004:1> puts "1..5"
irb(main):005:1> when 6..10
irb(main):006:1> puts "6..10"
irb(main):007:1> end
6..10
=> nil
irb(main):008:0>
```

6. Estruturas de Controle – While

- Semelhante ao while do C/C++ e Java.
- Forma:

while (teste lógico)

comandos

end

```
irb(main):016:0> def a( n )
irb(main):017:1> f = 0
irb(main):018:1> while ( f < n )
irb(main):019:2> puts f
irb(main):020:2> f += 1
irb(main):021:2> end
irb(main):022:1> end
=> nil
irb(main):023:0> a( 5 )
0
1
2
3
4
=> nil
```

6. Estruturas de Controle – For

- Semelhante ao for de Java.
- Forma:

for variavel in intervalo

comandos

end

```
irb(main):024:0> for num in (4..6)
irb(main):025:1> puts num
irb(main):026:1> end
4
5
6
=> 4..6
```

6. Estruturas de Controle – Until

- Semelhante ao do-while de C/C++.
- Forma:

begin

comandos

end until (teste logico)

Exemplo:

i = 10

begin

puts i

i = i – 1

end until (i == 0)

```
irb(main):001:0> i = 10
=> 10
irb(main):002:0> begin
irb(main):003:1* puts i
irb(main):004:1> i = i - 1
irb(main):005:1> end until ( i == 0 )
10
9
8
7
6
5
4
3
2
1
=> nil
irb(main):006:0>
```

7. Métodos

- As funções, ou métodos como são chamados em Ruby, são definidos seguindo a forma:

def nome_da_função(parametro1, parametro2, ...)

<bloco de comandos>

end

Exemplo:

```
def dict(a, b)
    return a
    puts b
end
```

```
irb(main):012:0> def soma( a, b )
irb(main):013:1> puts a + b
irb(main):014:1> end
=> nil
irb(main):015:0> soma( 10, 20 )
30
=> nil
```

8. Estruturas de Dados

- Existem apenas duas estruturas de dados pré-definidas junto a linguagem. Arrays e Hash

8. Estruturas de Dados – Array

- Arrays são equivalentes aos vetores de C/C++, com exceção de serem dinamicamente alocadas e podem ter vários tipos simultaneamente.

```
irb(main):016:0> ary = [ 1, 2, "3" ]
=> [1, 2, "3"]
irb(main):017:0> ary + [ "foo", "bar" ]
=> [1, 2, "3", "foo", "bar"]
irb(main):018:0> ary * 2
=> [1, 2, "3", 1, 2, "3"]
irb(main):019:0> ary[ 0, 2 ]
=> [1, 2]
irb(main):020:0> str = ary.join( ":" )
=> "1:2:3"
irb(main):021:0> str.split( ":" )
=> ["1", "2", "3"]
irb(main):022:0>
```

8. Estruturas de Dados – Hash

- Hashs são dicionários em que cada elemento do “vetor” é associado a uma chave.

```
irb(main):022:0> h = { 1 => 2, "2" => "4" }
=> {1=>2, "2"=>"4"}
irb(main):023:0> h[ 1 ]
=> 2
irb(main):024:0> h[ "2" ]
=> "4"
irb(main):025:0> h[ 5 ]
=> nil
irb(main):026:0> h[ 5 ] = 10 # colocando um novo valor
=> 10
irb(main):027:0> h
=> {1=>2, "2"=>"4", 5=>10}
irb(main):028:0> h.delete 1 # deletando um valor atraves da chave
=> 2
irb(main):029:0> h[ 1 ]
=> nil
irb(main):030:0> h
=> {"2"=>"4", 5=>10}
irb(main):031:0>
```

9. Classes

- Ruby, por ser uma linguagem orientada a objetos, possui também o conceito de classes. A definição de classe segue a regra:

class Nome_da_classe

definições

end

```ruby
# Teste.rb

class Exemplo
        def initialize( v = 0 )
                @valor = v
        end

        def imprimeValor
                puts @valor
        end
end

a = Exemplo.new( 10 )
a.imprimeValor
```

```
ncd@ncd-virtual-machine ~/Área de Trabalho $ ruby teste.rb
10
```

9. Classes

- Métodos e atributos, semelhantemente a Java e C++ podem ser definidos como “public”, “private” e “protected”. Por *defualt,* métodos são públicos e atributos privados.

9. Classes – Atributer readers e writers

- Ruby provém geradores automáticos de getters e setters em tempo de execução para os atributos com os comandos **attr_writer: var** para setters e **attr_reader: var** para getters. Existe ainda o **attr_accessor: var** que gera getters e setters

```ruby
# Teste.rb

class Exemplo

   attr_accessor :valor

   def initialize( v = 0 )
      @valor = v
   end

   def imprimeValor
      puts @valor
   end
end

a = Exemplo.new( 20 )
puts a.valor
```

```
ncd@ncd-virtual-machine ~/Área de Trabalho $ ruby teste.rb
20
```

9. Classes – Herança

- Ruby também possui o mecanismo de herança. A sintaxe:

class Nome_da _classe <nome_da_classe_pai

comandos

end

```ruby
# Teste.rb

class Exemplo

   attr_accessor :valor

   def initialize( v = 0 )
      @valor = v
   end

   def imprimeValor
      puts @valor
   end
end

class Filho < Exemplo

   attr_accessor :nome

   def initialize( v = 0, n = "Vitor" )
      super( v )
      @nome = n
   end
end


a = Filho.new( 20 )
puts a.valor
puts a.nome
```

9. Classes – Herança

```
ncd@ncd-virtual-machine ~/Area de Trabalho $ ruby teste.rb
20
Vitor
```

9. Classes – Herança Múltipla

- Ruby não suporta herança múltipla, ao invés disso usa o recurso de mixins para emular a herança múltipla.
- **class Pessoa < Mamifero # Herança de Mamifero include Humano # Emulando herança múltipla**
- **end**

10. Polimorfismo – Coerção

```
irb(main):001:0> i = 2
=> 2
irb(main):002:0> f = 2.5
=> 2.5
irb(main):003:0> a = i + f
=> 4.5
```

10. Polimorfismo – Sobrecarga

```ruby
# Sobreescrita.rb

class Teste

   attr_accessor :x, :y

   def initialize( a = 0, b = 0 )
      @x = a
      @y = b
   end

   def ==(right)
      ( @x == right.x ) && ( @y == right.y )
   end
end

a = Teste.new( 1, 2 )
b = Teste.new( 0, 2 )
c = Teste.new( 1, 2 )
puts a == b
puts c == a
```

```
administrador@administrador-virtual-machine ~/Área de Trabalho $ ruby SobreEscri
ta.rb
false
true
```

10. Polimorfismo – Paramétrico

- Pode-se dizer que todas as funções de Ruby são usam o polimorfismo paramétrico, a não ser que a função contenha alguma operação que a impeça.

```
irb(main):016:0> def soma( a, b )
irb(main):017:1> puts a + b
irb(main):018:1> end
=> nil
irb(main):019:0> soma( 1, 2 )
3
=> nil
irb(main):020:0> soma( "Vi", "tor" )
Vitor
=> nil
irb(main):021:0>
```

10. Polimorfismo – Inclusão

```ruby
# Teste.rb

class Exemplo

   attr_accessor :valor

   def initialize( v = 0 )
      @valor = v
   end

   def imprimeValor
      puts @valor
   end
end

class Filho < Exemplo

   attr_accessor :nome

   def initialize( v = 0, n = "Vitor" )
      super( v )
      @nome = n
   end
end


a = Filho.new( 20 )
puts a.valor
puts a.nome
```

```
ncd@ncd-virtual-machine ~/Area de Trabalho $ ruby teste.rb
20
Vitor
```

11. Tratamento de Exceção

- Ruby suporta tratamento de exceções por meio da sintaxe:
- **begin**
  - **Código**

  rescue
  - **Tratamento da exceção**

```
irb(main):021:0> def divide( a, b )
irb(main):022:1> begin
irb(main):023:2* puts a / b
irb(main):024:2> rescue
irb(main):025:2> puts "Erro: Divisao por zero\n"
irb(main):026:2> end
irb(main):027:1> end
```

```
irb(main):030:0> divide( 2, 0 )
Erro: Divisao por zero
```

11. Tratamento de Exceção

- Outros comandos para tratamento de exceções são:
  - fail // semelhante ao throw do Java
  - retry // Usado dentro do bloco rescue. Ele diz ao interpretador para novamente executar o bloco dentro de begin após o tratamento de exceção.
  - ensure (código) // Comando que executa independentemente do sucesso ou fracasso dentro de begin.

12. Concorrência

- Ruby possui suporte a concorrência por meio da classe Threads e o uso de semáforos por meio da classe Mutex.

13. Ruby Gems

- Gerenciador de pacotes de Ruby.
- Disponibiliza um formato padrão para a distribuição de programas e bibiliotecas Ruby, são os pacotes denominados “gems”.
- Tem como fonte padrão o seu site oficial (rubygems.org), no site há espaço para desenvolvedores cadastrados disponibilizarem suas bibiliotecas para toda comunidade.

14. Avaliação da Linguagem

Aplicabilidade	Simulações, Robótica, Redes, Telefonia, Administração de Sistemas, Aplicações Web
Concorrência	Sim (threds, semaforos)
Confiabilidade	Média
Custo	Baixo (Linguagem free, altamente portavel)
Eficiência	Baixa (Interpretada)
Excessões	Possui tratamento, a cargo do programador
Facilidade de Aprendizado	Boa
Gerencia de Memória	Gabarge Colector
Integração	Java, C#, PHP

14. Avaliação da Linguagem

Legibilidade	Boa
Método de Projeto	Interpretada, multiparadigma (funcional, OO, imperativa, reflexiva)
Passagem de parâmetros	Por cópia da referência
Polimorfismo	Herança simples. Coerção. Não admite sobrecarga
Portabilidade	Alta (Linux, windows, Mac, etc)
Redigibilidade	Excelente
Reusabilidade	Alta
Ortogonalidade	Ruim
Verificação de Tipos	**Tipagem dinâmica e forte**

15. Presença de Ruby no mercado

- Ruby e suas tecnologias são utilizadas por grandes corporações em diversos tipos de aplicações, destinadas por exemplo, a telefonia, gerência de sistemas e redes, portais corporativos, comércio eletrônico, redes sociais e até mesmo robótica.

15. Presença de Ruby no mercado

- Segundo informação do site Workingwithrails, grandes empresas brasileiras como Rede Globo, Grupo Abril, Locaweb, e gigantes internacionais como BBC, AOL, Amazon, Groupon entre outras utilizam a tecnologia Ruby on Rails em suas organizações.

15. Presença de Ruby no mercado

Histórias de Sucesso

- *Simulação*
  - *NASA Langley Research Center*
- *Negocio*
  - Toronto Rehab
- *Robótica*
  - MORPH
- *Redes*
  - Open Domain Server
- *Telefonica*
  - Lucent
- *Administração de Sistemas*
  - Level 3 Communications
- *Aplicações Web*
  - Basecamp
  - 43 Things
  - A List Apart
  - Blue Sequence

16. Ruby on Rails

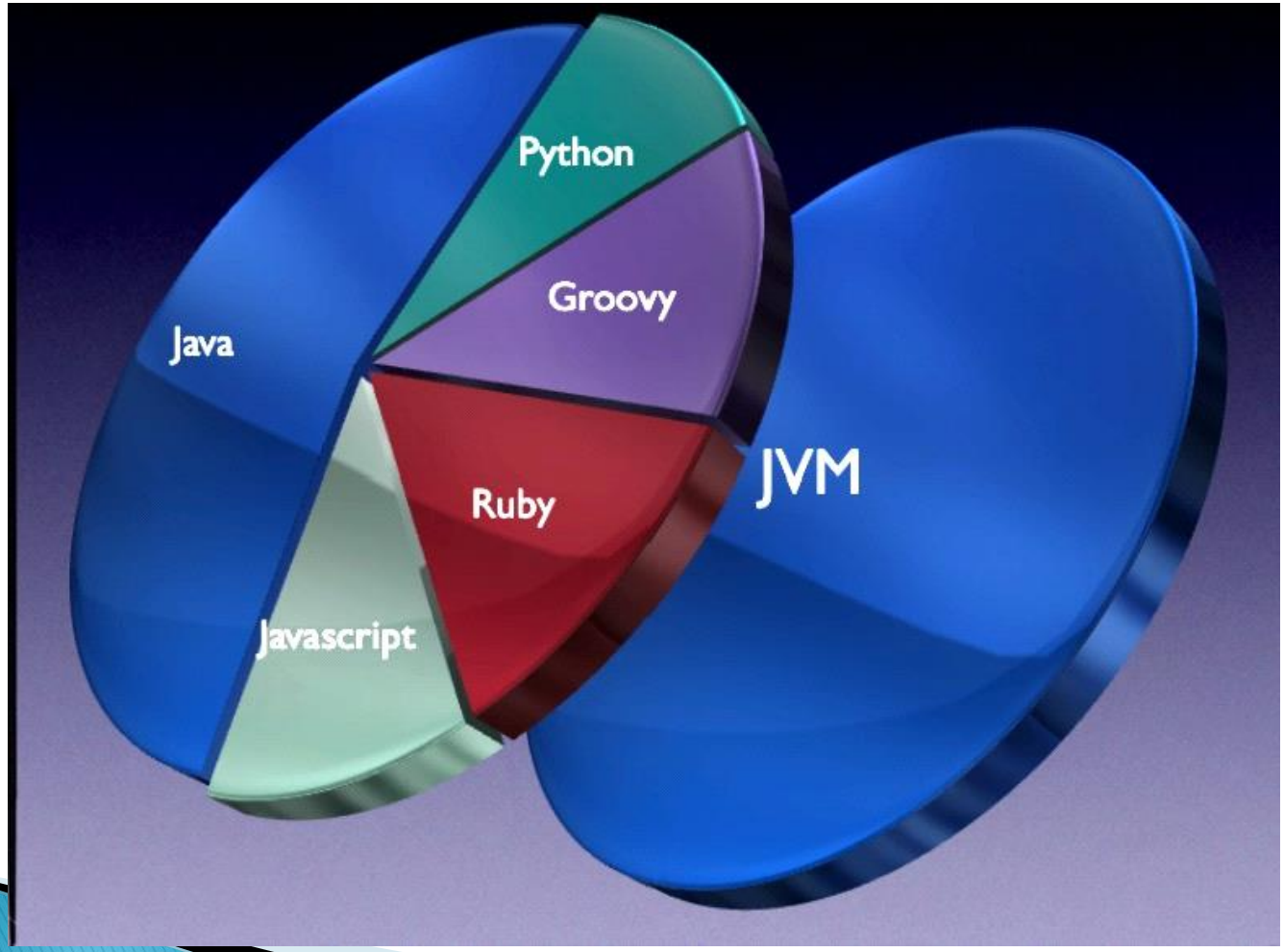
- Framework livre de desenvolvimento web. Permite desenvolvimento rápido e fácil de aplicações, seguindo o padrão de arquitetura MVC (Model-View-Controller). Tem sido a maior razão para o sucesso de Ruby na web.

17. JRuby

- Implementação da linguagem Ruby para plataforma Java
-
  - Roda sobre a JVM
  - Threads Nativas
  - JIT (Just-in-time compiler)
  - Garbage colector

  - Possibilidade de utilizar bibliotecas e classes Java
  - Jar's, servlets, hibernate, JDBC, etc.
  - Suporta Rubygems
  - Suporta Rails

18. Utilização de Java como plataforma



Referências

- http://www.dotlib.com.br/i/4768a22a0d58da3cfa71294e40ed6229.pdf
- http://www.ruby-lang.org
- http://pt.wikipedia.org/wiki/Ruby_(linguagem_de_programa%C3%A7%C3%A3o)
- http://www.tryruby.org
- http://www.slideshare.net/Belighted/ruby-vs-java
- http://www.urubatan.com.br/ruby-on-rails-x-java-web/
- http://www.pardontheinformation.com/2008/09/java-vs-ruby-on-rails-it-is-dead-heat.html
- http://jruby.org/
- http://rubyonrails.com.br
- http://rubygems.org/
- http://www.youtube.com/watch?v=5b3qd2VMLws
- http://www.oficinadanet.com.br/artigo/1072/ruby_o_que_e