
**Universidade Federal da Bahia**

**Instituto de Matemática**

**MATC84 - Laboratório de Programação Web**

# Ruby

**Salvador, 7 de Setembro de 2016.**

# Linguagem Ruby

Relatório desenvolvido sobre
a linguagem Ruby, orientado pelo docente
Alex Pereira  para a disciplina
MATC84 - Laboratório de programação Web.

*Componentes:*
*Ian Pierre*
*José Espinheira*
*Philippe Correia*

**SUMÁRIO**

## 1. A linguagem e Seu Histórico

Ruby é uma linguagem considerada por muitos como uma linguagem de altíssimo nível por trazer avanços no que diz respeito a programação. De fato, Ruby é uma linguagem bastante poderosa. Além de ser interpretada e de tipagem dinâmica e forte, o Ruby é uma linguagem multiparadigma. Além de suportar orientação a objetos, Ruby provê suporte a programação funcional, imperativa e reflexiva. Além disso, vem evoluindo seu gerenciamento de memória de maneira bastante expressiva e veremos, no decorrer desse texto, detalhes sobre esse gerenciamento de memória. O Ruby foi planejada por Yukihiro “Matz” Masumoto em 1995. A ideia era ser uma linguagem de script e, segundo seu próprio criador, veio para juntar e melhorar qualidades de duas outras grandes linguagens. Para Yukihiro, a ideia de criar uma linguagem de script era que a linguagem fosse mais poderosa que Perl e mais orientada a objetos do que Python. E, já adiantando, ele conseguiu atingir esse objetivo de maneira bastante contundente. Ruby é uma linguagem poderosa onde tudo é objeto, inclusive os tipos básicos de dados (ints, floats, Strings e etc). Portanto não é por menos que Ruby figura entre uma das dez linguagens de programação mais populares do mundo, segundo o Índice Tiobe.

Como dito no tópico anterior, a linguagem Ruby foi originalmente desenvolvida em 1995, porém sua concepção aconteceu em fevereiro de 1993 com a ideia de Yukihiro criar uma linguagem que balanceasse programação funcional e imperativa. O lançamento do Ruby só aconteceu em 1999, quando foi apresentada ao público na sua versão 1.3 e já em 2000 a linguagem recebeu seu primeiro livro dedicado em inglês (Programming Ruby) e começou a crescer fora do Japão. O livro, inclusive, for liberado gratuitamente para o público e isso ajudou muito no processo de adoção da linguagem por pessoas que tinha a habilidade de ler em inglês.

Por volta de 2003 houve um grande aumento no uso da linguagem pois o framework Ruby on Rails foi lançado. Esse framework é escrito em Ruby e se tornou tão popular que os menos familiarizados com a linguagem pensa que Ruby on Rails é o nome da linguagem. Hoje em dia, Ruby on Rails é o que vem a mente da grande maioria dos desenvolvedores quando o assunto é Ruby e, ainda hoje, é responsável pelo crescimento constante da linguagem. Hoje o Ruby está na sua versão 2.3.1.

## 2- Onde o Ruby se Aplica

Ruby é uma linguagem bastante versátil sendo possível desenvolver vários tipos de aplicações usando essa linguagem. Alguns tipos de aplicações são mais difundidos e tem uma disseminação de conteúdo e framework maior, como é o caso de Web. Mas também temos framework de mobile voltado para Ruby, como é o caso da RubyMotion.

A força do Ruby está, obviamente, na Web, é lá que alinhado ao Ruby om Rails faz sucesso entre os programadores, principalmente os que trabalham nas start-ups. A força do Ruby na web agregado ao Ruby on Rails é tão grande que vários programadores pensam que Ruby on Rails é a linguagem.

No entanto, temos boas iniciativas no que diz respeito ao Ruby e dispositivos móveis como é o caso do RubyMotion, um framework para aplicações *cross-platform mobile.* Existem ainda alguns outros bons frameworks mobile para o Ruby como é o caso do Mruby, criado pelo próprio criador do Ruby  e o Ruboto.

Durante nossas pesquisas, verificamos que é possível mas não necessariamente prático desenvolver outras vertentes de aplicações usando Ruby. Foi possível encontrar aplicações cliente-servidor com sockets e até jogos mais leves. No caso de jogos, é pouco aconselhável o desenvolvimento em Ruby, pelo menos se o jogo não for um projeto pequeno. Os desenvolvedores atestam que Ruby tem performance muito baixa para jogos.

## 3. Quem Usa

### Codeminer (Brasil)

Codeminer é uma das empresas pioneiras na utilização de Ruby para projetos empresariais. Ação tida como corajosa por uns e precipitadas por outro. No entanto, o sucesso da empresa é inegável e a figura do seu líder, Fábio Akita, extrapola a atmosfera da empresa. Famoso por suas palestras sobre O Ruby e o Ruby on Rails, Akita é figura conhecida pelos desenvolvedores de Ruby. Fábio Akita, inclusive, já participou como palestrante na Semana de Computação - UFBA 2013 (SEMCOMP).

A empresa utiliza o framework Ruby on Rails para construção de projetos de clientes e prestação de consultoria para projetos web com a mesma tecnologia.

### Bandcamp (E.U.A.)

Empresa de venda de música online e promoção de artistas. Com sua versão web baseada em Rails, a empresa passou na utilização do GenyMotion para levar um novo aplicativo para a plataforma IOS de forma nativa, em pouco tempo e utilizando a mesma equipe de desenvolvimento de software.

**Basecamp (E.U.A)**

Empresa focada em gerenciamento de projetos, cuja principal produto (Basecamp.com) utiliza o framework Ruby on Rails em todo sistema web. O produto conta com diversas ferramentas para mensuração e identificação de tarefas, assim como manipulação de arquivos, definições de metas e afins.

## 4. Detalhes do Ruby

Ruby é uma linguagem interpretada e de tipagem forte e dinâmica, como dito no iníco deste texto. Portanto, para executar algum programa nessa linguagem é necessário a instalação prévia de algum interpretador em sua máquina.

### 4.1 Interpretador

Como todo início, os programadores da linguagem Ruby não tinham muita opção no que diz respeito a interpretador. A única opção de interpretador disponível era o interpretador criado pelo próprio Yukihiro Matsumoto, criador da linguagem. Esse interpretador foi escrito em C e era bem simples e não trazia grande complexidade no tratamento de memória, por exemplo. Hoje em dia, interpretadores Ruby já se baseia em uma máquina virtual com recursos mais avançados e complexos.

Temos várias maneiras de instalação dos pacotes necessários para utilização do Ruby. Além do download no site oficial, temos as soluções prontas para instalação do Ruby em diversas plataformas através de métodos *“one-click-installer”.*

A grande maioria das distribuições Linux possuem o pacote de uma das últimas versões estáveis pronto para ser instalado. O exemplo mais comum é o de instalação para o Ubuntu:

```
sudo apt-get install ruby2.0 ruby2.0-dev build-essential libssl-dev zlib1g-dev \ruby-switch
```

O comando acima instala o Ruby e alguns pacotes necessários para instalar gems que contenham código C.

### 4.2 Gerenciamento de memória

Apesar de estar ligado fortemente ao C desde a sua origem, o Ruby trata o gerenciamento de memória de maneira bem diferente do C, se aproximando mais do Java e C# nesse quesito.

Desde a sua concepção, o Ruby utiliza o famoso *Garbage Collector* para gerenciar a memória de maneira automática. Uma das implementações bem difundidas do garbage collector é o algoritmo *mark-and-sweep* (marca e varre). Basicamente, o garbage collector percorre a memória heap buscando os objetos referenciados e marcando-os, após essa fase de marcação, o garbage collector varre os espaços que não foram marcados, liberando, assim memória para a aplicação. O Ruby, ao ser executado, aloca uma porção de memória e subdivide essa alocação em slots de 40 bytes. O Ruby só requisita memória ao kernel caso, mesmo após o trabalho do garbage collector, não tenha mais slots para ser usado.

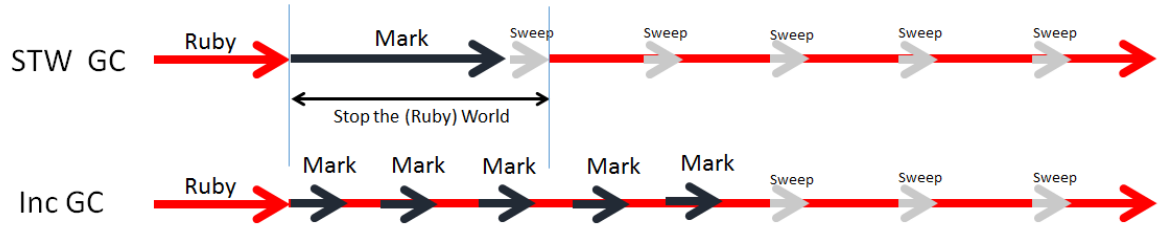
Um problema nessa abordagem e que muitas outras linguagens enfrentavam é que o garbage collector demorava muito tempo na marcação e isso fez com que o problema fosse batizado de “Stop the World”. Maneiras vinham sendo discutidas e uma implementação foi adotada no Ruby 2.1, a solução foi batizada de Generational Garbage Collector. Essa implementação traz consigo a ideia de separar a memória heap em dois espaços: um dedicado aos objetos novos e um dedicado aos objetos velhos. A base que justifica essa implementação é a máxima que diz que a grande maioria dos objetos morrem cedo, e isso é verdade se pensarmos em variáveis de métodos, por exemplo. Se um objeto “sobreviver” a fase de marcação três vezes, esse objeto é promovido à velha geração. Acontece que o Ruby concentra esforços na verificação e marcação dos objetos novos na maioria das vezes, fazendo com que a verificação aconteça de maneira mais rápida na maioria dos casos.

Apesar de minimizar o problema do “Stop the World” ainda temos verificações ocasionais nos objetos velhos que ainda causa a parada, mesmo que um número menor de vezes em relação ao algoritmo anterior. Pensando nisso, o Ruby não demorou em melhorar ainda mais seu garbage collector e já na versão 2.2 lançou uma adição ao garbage collector incremental que foi batizada de Incremental garbage collector.

Em linhas gerais, essa ideia é de não mais deixar a fase de marcação ser executada até finalizar de uma única, ao invés disso, no garbage collector incremental a fase de marcação acontece em vários espaços de tempo menores. Obviamente, o tempo total de marcação é um pouco maior por conta do overhead, mas o ganho de ter pausas menores justifica o overhead. Além dessa melhoria, temos a introdução do algoritmo de Tricolor Mapping que traz a ideia de marcações em branco, cinza e preto para que o processo de coleta de lixo seja ainda mais eficaz. Nessa solução todos os objetos existentes são marcados de branco e os objetos que estão realmente vivos são marcados de cinza. Depois um objeto cinza é escolhido e todos os objetos que aquele objeto cinza referencia é visitado e marcado de cinza, depois de todas referências estarem marcadas de cinza, a primeira referência, ou referência pai, é marcada de preto. Esse processo se repete até que não existam mais objetos cinza. Depois, como pode ser observado, basta coletar os objetos marcados da cor branca pois todos os objetos vivos estão marcados de preto. A parte incremental acontece pois o algoritmo visita um objeto marcado de cinza e faz o processo de marcar os referenciados por ele de cinza, depois volta para a execução da aplicação. Depois de um tempo, pega outro objeto cinza e faz o mesmo processo.

Essa solução traz uma melhora de performance significativa como pode ser observado abaixo:

### 4.3 RubyGems




O Ruby não fica atrás no que diz respeito a gerencia de pacotes. No Ruby temos o RubyGems. O RubyGems é um gerenciador de pacotes flexível, eficiente e bastante avançado. E aqui vale a mesma regra das outras linguagens: Enquanto desenvolvemos, percebemos que várias funcionalidades já foram implementadas por outro desenvolvedor e, como desenvolvedores, sabemos que não precisamos reinventar a roda e portanto essas bibliotecas são disponibilizadas e podem ser baixadas pelo RubyGems. Nesse caso, essas bibliotecas recebem o nome de gems.

Para gerenciar as dependências, podemos utilizar a gem Bundler. Com o Bundler no seu projeto, basta adicionar as dependências do projeto do Gemfile e executar o comando “bundle” e o Bundler se encarregará de tratar as gems que o nosso projeto necessita.

## 4.4 Implementações alternativas

O Ruby vem crescendo e se popularizando cada vez mais, e essa popularização trouxe consigo, principalmente após o surgimento do Ruby on Rails, algumas implementações alternativas da linguagem. A maioria delas segue uma tendência natural de serem baseados em uma Máquina Virtual em vez de serem interpretadores simples. Algumas implementações possuem até compiladores completos, que transformam o código Ruby em alguma linguagem intermediária a ser interpretada por uma máquina virtual.

A principal vantagem das máquinas virtuais é facilitar o suporte em diferentes plataformas. Além disso, ter código intermediário permite otimização do código em tempo de execução, feito através da **JIT**.

### 4.4.1 Jruby

O Jruby é referenciado como a primeira implementação alternativa completa da versão 1.8.6 do Ruby e é também a principal implementação da linguagem Java para JVM.

O fato de rodar na JVM garante uma melhora em relação a um simples interpretador. Com essa implementação rodando na JVM, é possível operar nos modos de compilação Ahead of Time e o Just in Time. Entretanto, é possível operar no modo interpretador tradicional Tree Walker.

As vantagens são ÓbvIas: Interoperabilidade com código Java existente e aproveitamento das funcionalidades de uma das plataformas de execução de código mais difundidas e poderosas do mundo.

## 4.5 Executando códigos em Ruby

Diferente de outras linguagens, o Ruby tem um console onde os comandos em Ruby são inseridos, interpretados e retornados interativamente. O nome desse console é IRB (Interactive Ruby Shell). O IRB avalia cada linha inserida e já mostra o resultado imediatamente.

Além disso, é possível fazer um arquivo com o código e salvar com a extensão .rb e executa com, por exemplo, “ruby teste.rb”.

### 4.6 Impressão de Mensagens

Para fazer a impressão na linguagem Ruby temos três opções: puts, print e p. Portanto, é perfeitamente aceitável fazer um “Olá Mundo!” com:

```
puts "Olá Mundo"
print "Olá Mundo"
p "Olá Mundo"
```

A diferença é que O puts imprime o conteúdo e pula uma linha, o print imprime, mas não pula linha, já o p chama o método inspect do elemento.

### 4.7 Comentários

Para comentar em apenas uma linha é necessário usar “#”

```
# Imprime uma mensagem
puts "Olá mundo"
```

Ou comentários de blocos:

```
=begin
  Imprime uma mensagem
É só um olá mundo
=end
puts "Olá mundo"
```

Os comentários em bloco não são muito utilizados.

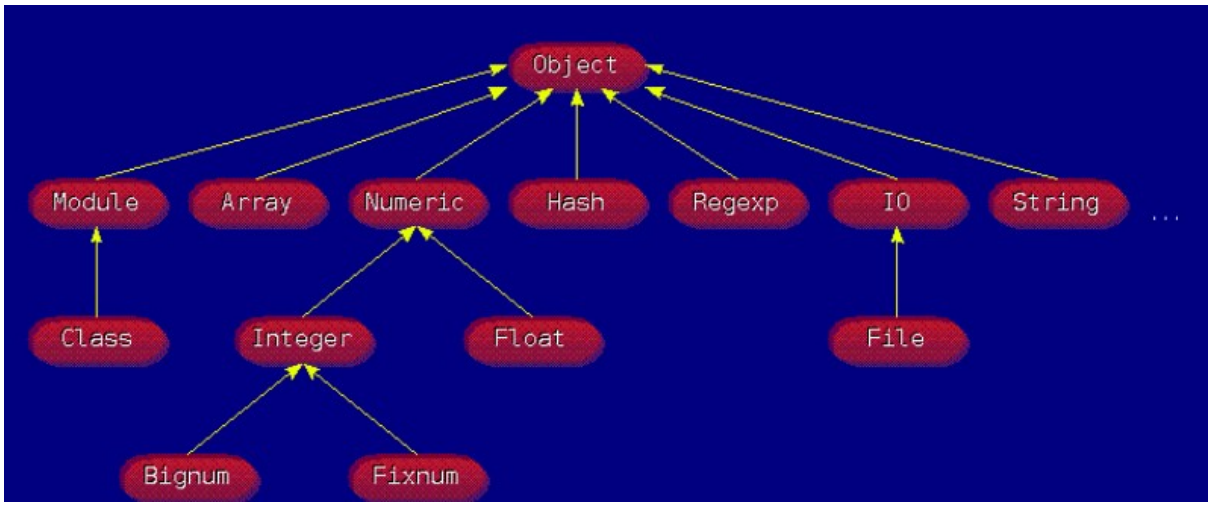
### 4.8 Algumas Palavras Reservadas

Segue uma lista de algumas palavras reservadas do Ruby

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| alias | 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 | | | |

### 4.9 Hierarquia de classes Ruby

Prezando pela orientação a objetos, tudo em Ruby é objeto. Não existem tipos primitivos, apenas objetos. De maneira análoga ao Java, todas as classes do Ruby herdam da classe Object. Podemos notar na imagem abaixo essa hierarquia. É importante salientar a classe Numeric, filha de Object. Essa classe é responsável por todos valores numéricos no Ruby. Ainda na mesma linhagem, temos Integer e Float que são filhas de Numeric. Float é utilizada para representar pontos flutuantes, já a classe Integer é abstrata e não pode ser instanciada, ao invés disso, temos duas classes filhas de Integer para representar inteiros, elas são Bignum e Fixnum, veremos mais dessas classes a seguir.




### 4.10 Valores numéricos

Como dito no tópico anterior, os valores numéricos no Ruby obedecem a hierarquia das classes e temos como principal diferença em relação às outras linguagens a diferença no nome de int para Fixnum para inteiros pequenos e  de Long para Bignum para inteiros grandes. Além do Fixnum e do Bignum temos o Float, que representa ponto flutuante.

```
idade = 14 <= fixnum;
```

```
idade = 999999999999999999999999 <= bignum;
idade = 12.3 <= float.
```

Operações matemáticas básicas

- + Soma
- - Subtração
- / Divisão
- * Multiplicação
- ** Potência
- % Modulo (retorna o resto de uma divisão)

### 4.11 Variáveis e Atribuições

A tipagem em Ruby é dinamicamente e fortemente tipada. Isso veremos um pouco melhor mais adiante. Agora, precisamos saber que a atribuição de variável em Ruby funciona da mesma maneira que Java e C, usando “=”. No entanto, para criar uma variável, é necessário apenas dizer o nome dela e atribuir o valor, como por exemplo:

```
idade = 150
```

Isso é conhecido como inferência de tipos ou tipagem implícita e é algo comum em linguagens consideradas de altíssimo nível que prezam pela agilidade no desenvolvimento, como é o caso do Ruby e Python, por exemplo. O interpretador infere automaticamente durante a execução do código. Nesse caso, essa variável é do tipo Fixnum, que representa os números inteiros pequenos.

### 4.12 Tipagem

Como dito anteriormente, os tipos em Ruby na verdade são objetos, ou seja: não existem tipos primitivos e sim objetos que pertencem à classes. Como disse anteriormente, Ruby é dinamicamente e fortemente tipada. Vejamos o que cada um destes adjetivos, de fato, significam:

#### 4.12.1 Dinamicamente tipada

Quer dizer que, a cada interação, o tipo é verificado. Caso ele mude, o Ruby se encarrega de mudar a vinculação de tipo daquela variável. Isso traz dinamicidade, apesar de programadores mais puristas acharem que isso deixa o programa confuso e com pouco controle das variáveis. No código a seguir fica bem claro:

```
x = 100
```

```
(1..4).each{ x = x * 100 ;  puts "#{ x.class} #{x}" } //o método each será explicado na seção destinada à Arrays.
```

Que gera o seguinte resultado após todas as interações que o range nos proporciona:

```
Fixnum 10000
Fixnum 1000000
Fixnum 100000000
Bignum 10000000000
```

Como podemos ver a variável x começa como Fixnum e conforme aumenta passa para Bignum sem a necessidade de uma conversão.

Essa característica gera vários pontos positivos na linguagem, tais como:

- Diminui a verbosidade, pois não há a necessidade de fazer conversões;
- Tende a facilitar a vida do programador pois não há a preocupação com conversões.

Mas neste contexto nem tudo são flores, vejamos os pontos negativos:

- A linguagem tende a ser mais lenta, em tempo de execução, devido ao fato de que, a cada interação, o tipo ser verificado;
- Pode confundir o programador pois o mesmo pode não saber exatamente com quais tipos está tratando.

Outro exemplo:

```
idade = 150
idade = “150”
```

A variável idade mudou de inteiro para string durante a execução.

Para descobrir o tipo da variável em Ruby basta usar .class (pois toda variável é um objeto em Ruby).

### 4.12.2 Fortemente Tipada

Este é um conceito confuso, mas uma boa forma de explicá-lo é dizendo que todas as variáveis devem ter um tipo - ou seja: fazer parte de uma classe no caso do Ruby - e que cada tipo segue a risca seu contrato. Muitas pessoas ligam a ideia de dinamicamente tipada à fracamente tipada e isso não é verdade, como no caso do Ruby. O ruby, diferente do Javascript não permite, por exemplo, concatenação de uma letra com um número. Por exemplo:

```
a = 100
b = "Ruby on Rails"
a + b
TypeError: String can't be coerced into Fixnum
```

Como é possível notar, em Ruby não podemos somar um String com Fixnum pois essa operação não está no contrato da classe String e Fixnum.

### 4.13 Strings

A string em Ruby pode ser declarada com aspas simples ou duplas e podem ser mudadas, diferentes de Java onde as Strings são imutáveis. O problema de aspas simples é que ela não permite interpolação de strings.

Portanto:

```
mensagem = “bom dia”
```

Pode ser mudado mais tarde no programa para:

```
mensagem = ‘boa tarde’
```

Para concatenar strings, podemos usar o operador <<. Por exemplo:

```
mensagem << “, tudo bem¿”
```

Vai retornar:

```
“boa tarde, tudo bem¿”
```

Obviamente, existem algumas outras maneiras de concatenar strings em Ruby.

A interpolação de String vem para prover ainda mais dinamicidade, praticidade e legibilidade de código no Ruby. No ruby é possível colocar as variáveis dentro de uma String, como podemos ver a seguir:

```
rb(main):001:0> mensagem = "bom dia, "
=> "bom dia, "
irb(main):002:0> aluno = "Ian Pierre"
=> "Ian Pierre"
irb(main):003:0> "Então, #{mensagem} #{aluno}"
=> "Então, bom dia,  Ian Pierre"
```

### 4.14 Symbols

Os símbolos em Ruby é o que temos de mais próximo das Strings imutáveis. Para declarar um símbolo, bastar colocar dois pontos antes do mesmo:

```
:symbol
```

Os símbolos são muito utilizados no Hash (explicado posteriormente, mas que basicamente é um array associativo) pois os símbolos são imutáveis e um símbolo é único na aplicação melhorando o uso da memória, por exemplo. Isso quer dizer que dois símbolos com o mesmo nome apontam para o mesmo espaço de memória, diferente das Strings.

### 4.15 Range

Ruby fornece uma maneira de trabalharmos com sequências de uma forma bem simples: (1..3) # range representando números de 1 a 3. ('a'..'z') # range representando letras minúsculas do alfabeto (0...5) # range representando números de 0 a 4. Perceba que ao utilizar dois pontos, o range é criar levando em consideração o último elemento, já usando três prontos o range é considerado apenas até o penúltimo item.

### 4.16 Booleanos

Os operadores booleanos aceitam quaisquer expressões aritméticas, como por exemplo:

```
3 > 2
true
3+4-2 <= 3*2/4
False
```

Os operadores booleanos são: ==, >, <, >= e <=. Expressões booleanas podem ainda ser combinadas com os operadores “e” “ou”. Eles podem ser definidos no código como && ou and e || ou or.

### 4.17 Arrays

Os arrays no Ruby são bem parecidos com os arrays em outra linguagem, uma diferença bastant expressiva dos arrays no Ruby é que os mesmos podem conter objetos de tipos diferentes, como é o caso a seguir:

```
valores [“Pessoa”, 1, 2.56, 4565758768976978, “Azul Marinho”]
```

Isso reflete muito bem a filosofia do Ruby, pois essa qualidade nos traz dinamicidade e flexibilidade. Único problema é que, obviamente, não é possível utilizar os métodos da classe Array em arrays com valores de objetos diversos.

No entanto, caso tenhamos um array de um único tipo é possível fazer algumas operações interessante sobre o mesmo. Considere o exemplo do nosso array:

```
valor = [5, 2, 1, 4, 3]
```

podemos utilizar o método sort para ordenar o nosso array:

```
valor.sort
```

e o resultado seria:

```
valor [1, 2, 3, 4, 5]
```

Um outro método interessante é o “include?”. Esse método retorna true se o valor está no array e false se não está.

`valor.include?(8)` retornaria false

`valor.include?(1)` retornaria true.

A classe Array tem muitos outros métodos igualmente úteis e interessantes, é altamente recomendável ler a documentação da classe para um melhor aproveitamento da mesma.

### 4.18 Hash

No ruby, Hash é uma classe que permite a criação de arrays associativos. Um array associativo é um array onde cada valor tem uma chave para facilitar a eventual recuperação do mesmo.

```
tenta = ['um' => 1, 'dois' =>2]
```

### 4.19 Estruturas de Controle e Repetição

O if do ruby aceita qualquer expressão booleana, no entanto, cada objeto em Ruby possui um "valor booleano". Os únicos objetos de valor booleano false são o próprio false e o nil. Portanto, qualquer valor pode ser usado como argumento do if.

```
x=1
if x > 2
   puts "x é maior que 2"
elsif x <= 2 and x!=0
   puts "x é 1"
else
   puts "I can't guess the number"
end
```

Temos ainda o if ternário que já aparece no PHP:

```
numero = 2
if numero>0 ? puts “positivo” : puts “negativo”
```

Podemos também usar o if pós fixado, isso facilita a leitura por humanos. Portanto, esse if tem muito a ver com a filosofia do Ruby.

```
puts "eh numeros" if 30.class == Bignum
```

Além das estruturas descritas acima, temos uma estrutura no Ruby que é o UNLESS e funciona como uma negação do IF. Enquanto o executa caso a condição verdadeira, o unless executa caso a condição seja falsa.

```
unless true
      puts “equivalente a um if false”
else
      puts “essa sempre sera a saída nesse caso”
```

podemos usar o unless pós fixado, que promove uma legibilidade melhor:

```
puts “é uma string” unless “String”.class != String
```

Nesse caso, “é uma string” seria impresso na tela.

Temos também o switch:

```
def procura_sede_copa_do_mundo( ano )
     case ano
      when 1895..2005
           "Não lembro... :)"
      when 2006
           "Alemanha"
        when 2010
           "África do Sul"
        when 2014
           "Brasil"
    end
end
puts procura_sede_copa_do_mundo(1994)
```

O código acima funciona como uma série de if/elsif :

```
if 2006 == ano
  "Alemanha"
elsif 2010 == ano
  "África do Sul"
```

```
elsif 2014 == ano
  "Brasil"
elsif 1895..2005 == ano
      "Não lembro... :)"
end
```

Utilizar um laço de repetições pode poupar muito trabalho. Em ruby o código é bem direto inclusive usando o range:

```
for i in (1..3)
      x = i
end
```

O mesmo acontece com o while

```
i = 0
while i < 6
  i += 1
end
```

**4.20 Iteração**

Uma explicação rápida sobre iterações em arrays será dada nessa seção, para mais detalhes, é sempre importante consultar a documentação da linguagem.

Temos quatro formas de iterar sobre um array e a mais comum sem dúvidas é usando o método each. Usando esse método a lista será percorrida do começo ao fim.

```
      number = [1, 2, 3, 4, 5]
      numbers.each do |numero|
            puts numero
      end
```

o resultado é 1 2 3 4 5

Temos também o método select, esse método insere em um novo array os itens do array atual e pode fazer isso obedecendo uma regra, como no caso abaixo onde o array maior_que_dois está sendo criado e populado com os itens do array numbers que são maior que dois, nesse caso, 3, 4 e 5.

```
maior_que_dois = numbers.select do |number|

      numbers > 2

end
```

Temos ainda outros métodos de iteração que podem ser encontrados na documentação do Ruby.

Como é possível notar ao ver todos os trechos de código apresentados, o Ruby não faz uso de “;” ao final das linhas ou chaves. Todo controle de começo e fim de loops é feito por meio de “end” e indentação do código.

### 4.21 Classes, objetos, métodos e orientação à objeto em Ruby

#### 4.21.1 Orientação à objetos

No mundo das linguagens, muito se discute se quais as linguagens são puramente orientadas a objeto ou não. Java é uma exemplo de linguagem que causa grande contradição e polêmica. Muito se discute que muita coisa não se comporta como objetos. Ruby é considerada uma linguagem puramente orientada a objetos, já que **tudo** em Ruby é um objeto, inclusive as classes.

#### 4.21.2 Métodos

Para criar um método, é necessário iniciar com “def” e terminar com “end”. Obviamente, é possível receber parâmetros nesses métodos. Exemplo:

```
def pessoa.vai(lugar)
   "indo para " + lugar
end
```

Uma outra diferença de outras linguagens é que não declaramos o retorno do método. O Ruby retorna o resultado da execução da última linha do método. No caso acima, no caso acima, o retorno seria um String.

Para visualizar esse retorno funcionando, podemos acessar o método e imprimir o retorno do mesmo:

```
puts pessoa.vai("casa")
```

#### 4.21.3 Classes

A classe, de maneira análoga ao método, é apenas necessário definir a palavra reservada “class” no inicio e “end” no fim, sem o uso de chaves.

```
class Pessoa
  def novo_metodo
     # ...
  end
end
```

Um método pode invocar outro método do próprio objeto. Para isto, basta usar a referência especial self, que aponta para o para o próprio objeto. É análogo ao this de outras linguagens como Java e C#.

#### 4.21.4 Atributos em Ruby

Os atributos ou variáveis de instância em Ruby serão sempre privados, ou seja, não podem ser modificados fora da classe, somente um método do objeto é capaz de modificar um atributo do mesmo. Dessa forma, existe uma melhora inerente no encapsulamento. Todos esses atributos começam com “@”.

```
class Pessoa
  def muda_nome(novo_nome)
     @nome = novo_nome
  end

  def diz_nome
     "meu nome é #{@nome}"
  end
end
```

O primeiro seria análogo aos métodos set e o segundo ao get do java, por exemplo.

Existem vários outros detalhes técnicos interessantes na linguagem Ruby, mas vamos evitar nos estender, já que o objetivo é criar um interesse na linguagem e prover conhecimento básico.

## 5. Frameworks

### Ruby on Rails

**Ano de Lançamento:** 2004

**Plataforma:** Web (Padrão MVC)

**Utilizado Por:** Linkedin, Airbn, Github

Uma das tecnologias mais representante do modelo de negócio de startups. Focado na facilidade e velocidade de desenvolvimento de aplicações, o framework é adotado em ampla gama de empresas (Airbn, Ask.fm, Github ...) e se tornou referência para a linguagem.

Tem um de seus temas “convention over configuration” (convenções sobre configurações), o que é refletido em definições padrões serem seguidas ante ter implementações específicas para o projeto, facilitando a tomada de decisões e diminuindo a curva de aprendizagem.

Exemplo de um Hello World (pages_controller.rb):

```
class PagesController < ApplicationController
  def home
    puts "Honey, I'm home!"
  end
end
```

### Rubymotion

**Ano de Lançamento:** 2012

**Plataforma:** Mobile

**Utilizado Por:** Salesforce, Basecamp, Trought Works

Um projeto criado por um ex-funcionário da Apple, Laurent Sansonetti , responsável por uma implementação do Ruby para IOS e Android.

Adotado por empresas que já utilizam Ruby e portanto diminuem a curva de aprendizagem para lançar um aplicativo funcional.

Exemplo de um “Hello World” (main_activity.rb):

```
class MainActivity < Android::App::Activity
  def onCreate(savedInstanceState)
    super

    @text = Android::Widget::TextView.new(self)
    @text.text = 'Hello RubyMotion!'
    @text.textColor = Android::Graphics::Color::WHITE
    @text.textSize = 40.0
    self.contentView = @text
  end

  def dispatchTouchEvent(event)
    @counter ||= 0
    case event.action
      when Android::View::MotionEvent::ACTION_UP
        @counter += 1
        @text.text = "Touched #{@counter} times!"
        @text.backgroundColor = Android::Graphics::Color::BLACK
      when Android::View::MotionEvent::ACTION_MOVE
        @text.text = "ZOMG!"
        @text.backgroundColor = Android::Graphics::Color.rgb(rand(255), rand(255), rand(255))
      end
    true
  end
end
```

## 6. Referências

**Trabalho escrito desenvolvido por Ian Pierre, André, Rafael e Calebe na disciplina de Linguagem para aplicações comerciais serviu de base para o trabalho aqui desenvolvido.**


*Sobre o Ruby*

< https://www.ruby-lang.org/pt/about/ >

*Nuances do Ruby*

https://www.ruby-lang.org/pt/about/

https://www.caelum.com.br/apostila-ruby-on-rails/

*RubyMotion*

<http://www.rubymotion.com/>

*CodeShip Blog*

<https://blog.codeship.com/a-survey-of-non-rails-frameworks-in-ruby-cuba-sinatra-padrino-lotus/>

Bandcamp Success Storie

<http://www.rubymotion.com/references/success-stories/bandcamp>