

## CAPÍTULO 3

# Ruby básico

*"Pensa como pensam os sábios, mas fala como falam as pessoas simples"*  
— Aristóteles

## 3.1 - APRENDER RUBY?

Ruby on Rails é escrito em Ruby, e embora seja possível fazer aplicações inteiras sem ter um conhecimento razoável sobre Ruby, com o tempo surge a necessidade de entender mais profundamente determinado comportamento da linguagem e qual sua influência na maneira como o Rails trata determinada parte das nossas aplicações.

Conhecer bem a linguagem determinará a diferença entre *usar o Rails* e *conhecer o Rails*.

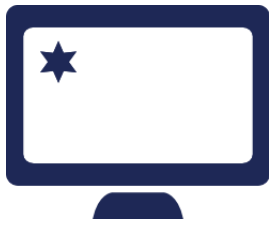
## 3.2 - EXECUTANDO CÓDIGO RUBY NO TERMINAL: IRB E ARQUIVOS .RB

O **IRB** é um dos principais recursos disponíveis aos programadores Ruby. Funciona como um console/terminal, e os comandos vão sendo interpretados ao mesmo tempo em que vão sendo inseridos, de forma interativa. O `irb` avalia cada linha inserida e já mostra o resultado imediatamente.

Todos os comandos apresentados neste capítulo podem ser executados dentro do `irb`, ou em arquivos `.rb`. Por exemplo, poderíamos criar um arquivo chamado **teste.rb** e executá-lo com o seguinte comando:

```
ruby teste.rb
```

Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](#)

### 3.3 – VARIÁVEIS, STRINGS E COMENTÁRIOS

Ao aprender uma nova linguagem de programação sempre nos deparamos com o famoso "Olá Mundo". Podemos imprimir uma mensagem com ruby utilizando pelo menos 3 opções: **puts**, **print** e **p**.

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

Mas qual a diferença entre eles? 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.

#### Comentários

Comentários em Ruby podem ser de uma linha apenas:

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

Ou comentários de blocos:

```
=begin
  Imprime uma mensagem
=end
puts "Oi mundo"
```

#### Comentários de bloco

Nos comentários de bloco, tanto o `"=begin"` quanto o `"=end"` devem ficar no início da linha, na coluna 0 do arquivo. Os comentários de blocos não costumam ser muito usados Ruby, a própria documentação do Rails usa apenas comentários de linha.

## Resultado da execução

Atenção para o modo de saída utilizado pelo Ruby e como poderemos identificar uma saída correta em nosso código:

- **Arquivo .rb:** Ao executarmos um código em um arquivo .rb, somente serão exibidas no terminal as saídas que nós especificamos com `puts` ou outro comando específico que veremos nos exemplos a seguir.
- **IRB:** Todo comando executado no IRB exibe ao menos uma linha que inicia com os caracteres `=>`. Essa linha indica o **retorno** da execução, como veremos nos exemplos a seguir. Caso utilizemos o `puts` no terminal, a execução resultará em mais de uma linha, sendo que a última, iniciando com `=>` ainda representa o retorno da operação. A linha indicando o retorno `nil` significa que o código executado não tem um valor de retorno definido. Nesses casos sempre obteremos o valor `nil` que é um valor equivalente ao `null` do Java ou outras linguagens.

## Palavras Reservadas do Ruby

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

## 3.4 – VARIÁVEIS E ATRIBUIÇÕES

Um dos mais conceitos básicos em linguagens de programação é a declaração de variáveis, que é apenas uma associação entre um nome e um valor. Em Ruby, basta definirmos o nome da variável e atribuir um valor usando o sinal `=`:

```
ano = 1950
```

Mas qual o tipo dessa variável? A variável `ano` é do tipo `Fixnum`, um tipo do Ruby que representa números inteiros. Mas não informamos isso na declaração da variável `ano`, isso porque na linguagem Ruby não é necessária esse tipo de informação, já que o interpretador da linguagem infere o tipo da variável automaticamente durante a execução do código. Esta característica é conhecida

como inferência de tipos.

## gets

O Ruby nos permite atribuir numa variável um valor digitado no console através do teclado, isso é feito através do método `%gets%`, mas com ele só poderemos atribuir `Strings%%`, veremos como atribuir outros tipos adiante. Exemplo:

```
irb(main):001:0> nome = gets
José
=> "José\n"
```

O `\n` representa um caracter de nova linha. Esse é um caracter que seu teclado manda quando é pressionada a tecla Enter.

## 3.5 - TIPAGEM

Por não ser necessária a declaração do tipo, muitos pensam que a linguagem Ruby é fracamente tipada e com isso o tipo não importaria para o interpretador. No Ruby os tipos são importantes sim para a linguagem, tornando-a fortemente tipada, além disso, o Ruby também é implicitamente e dinamicamente tipada, implicitamente tipada pois os tipos são inferidos pelo interpretador, não precisam ser declarados e dinamicamente tipada porque o Ruby permite que o tipo da variável possa ser alterado durante a execução do programa, exemplo:

```
idade = 27
idade = "27"
```

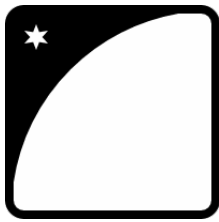
## O .class

Em Ruby temos a possibilidade de descobrir o tipo de uma variável utilizando o `.class`:

```
irb(main):001:0> num = 87
=> 87
irb(main):002:0> puts num.class
Fixnum
=> nil
```

**Você não está nessa página a toa**

Você chegou aqui porque a Caelum é referência nacional em cursos de Java,



Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

[Consulte as vantagens do curso \*Desenv. Ágil para Web com Ruby on Rails\*.](#)

### 3.6 – EXERCÍCIOS – VARIÁVEIS E ATRIBUIÇÕES

1. Vamos fazer alguns testes com o Ruby, escrevendo o código e executando no terminal. O arquivo utilizado se chamará **restaurante\_basico.rb**, pois nossa futura aplicação web será um sistema de qualificação de restaurantes.

- Crie um diretório no Desktop com o nome **ruby**
- Dentro deste diretório crie o arquivo **restaurante\_basico.rb**

Ao final de cada exercício você pode rodar o arquivo no terminal com o comando **ruby restaurante\_basico.rb**.

2. Vamos imprimir uma mensagem pedindo para o usuário digitar o nome do restaurante.

```
puts "Digite o nome do restaurante: "
```

3. Agora iremos guardar o nome do nosso restaurante em uma variável nome. Para isso usaremos o método **gets** para capturar o valor digitado no console.

```
puts "Digite o nome do restaurante"  
nome = gets
```

4. Por fim vamos imprimir o nome do nosso restaurante.

```
puts "Digite o nome do restaurante"  
nome = gets  
print "Nome do restaurante: "  
puts nome
```

### 3.7 – STRING

Um tipo muito importante nos programas Ruby são os objetos do tipo String. As Strings literais em Ruby podem ser delimitadas por aspas simples ou aspas duplas além de outras formas especiais que veremos mais adiante.

```
irb(main):001:0> mensagem = "Olá Mundo"
=> "Olá Mundo"
irb(main):002:0> mensagem = 'Olá Mundo'
=> "Olá Mundo"
```

## Mutabilidade

A principal característica das Strings em Ruby é que são mutáveis, diferente do Java, por exemplo.

```
irb(main):001:0> mensagem = "Bom dia, "
=> "Bom dia,"
irb(main):002:0> mensagem << " tudo bem?"
=> "Bom dia, tudo bem?"
irb(main):003:0> puts mensagem
Bom dia, tudo bem?
=> nil
```

O operador << é utilizado para a operação append de Strings, ou seja, para a concatenação de Strings em uma mesma instância. Já o operador + também concatena Strings mas não na mesma instância, isso quer dizer que o + gera novas Strings.

## Interpolação de Strings

Lembra da nossa mensagem que fizemos no último exercício?

```
#Declaração da variável %%nome%%
print "Nome do restaurante: "
puts nome
```

Nós podemos simplificá-la, uma alternativa mais interessante para criar Strings com valor dinâmico é a interpolação:

```
#Declaração da variável %%nome%%
print "Nome do restaurante: #{nome}"
```

Basta colocar dentro da String a variável entre as chaves, em #{ }. Mas fique atento, com Strings definidas com aspas simples não é possível fazer uso da interpolação, por isso prefira sempre o uso de String com aspas duplas.

Prefira sempre a interpolação ao invés da concatenação (+) ou do append (<<). É mais elegante e mais rápido.

## O método capitalize

Repare que no caso do restaurante o usuário pode digitar qualquer nome,

começando com ou sem letra maiúscula. Mas seria elegante um restaurante com nome que comece com letra minúscula? Para isso temos um método chamado **capitalize**, sua função nada mais é do que colocar a primeira letra de uma `String` em caixa alta.

```
irb(main):001:0> nome = "fasano"
=> "fasano"
irb(main):002:0> puts nome.capitalize
Fasano
=> nil
```

A chamada do método **capitalize** ao invés de alterar a variável para guardar o valor “Fasano”, retorna uma nova `String` com o valor com a primeira letra em caixa alta. Você pode confirmar este comportamento imprimindo novamente o nome, logo após a chamada do método **capitalize**, repare que foi impresso exatamente o mesmo valor que definimos na declaração da variável.

```
nome = "fasano"
puts nome.capitalize #Fasano
puts nome #fasano
```

O método **capitalize** é uma função pura, porque não importa quantas vezes seja invocado, retornará sempre o mesmo valor e também não causa efeitos colaterais, ou seja, não altera um valor já calculado anteriormente.

## Os métodos terminados em bang(!)

Mas e se quisermos que a mudança realizada pelo `capitalize` seja definitiva? Para isso acrescentamos o caracter **!** no final do método.

```
nome = "fasano"
puts nome.capitalize! #Fasano
puts nome #Fasano
```

O caracter **!** é chamado de bang e deve ser usado com moderação, já mudar definitivamente o valor onde foi utilizado.

## 3.8 – EXERCÍCIOS – STRINGS

1. Primeiramente, vamos alterar o modo como o nome do nosso restaurante é impresso. Abra o arquivo **restaurante\_basico.rb** e passe a utilizar a concatenação.

```
print "Nome do restaurante: " + nome
```

2. Já deu uma melhorada, só que aprendemos uma maneira mais elegante de fazer concatenação, que é a interpolação, vamos passar a utilizá-la.

```
print "Nome do restaurante: #{nome}"
```

## Default encoding

A partir de sua versão 1.9 o Ruby tem a capacidade de tratar de modo separado o encoding de arquivo (external encoding), encoding de conteúdo de arquivo (internal encoding), além do encoding aceito no IRB.

O encoding padrão a partir da versão 2.0 é o UTF-8 (Unicode), não sendo mais necessário configurar o encoding para utilização de acentuação nos caracteres.

Até a versão 1.9.3, caso fosse necessário utilizar o UTF-8, era preciso adicionar uma linha com um comentário especial na **primeira linha** dos arquivos **.rb**:

```
#coding:utf-8  
puts "Olá, mundo!"
```

Para o IRB podemos configurar o encoding no comando `irb` com a opção de encoding. O comando completo é `irb -E UTF-8:UTF-8`. Em algumas plataformas é possível utilizar a forma abreviada `irb -U`.

## Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.


Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](#)

## 3.9 – TIPOS E OPERAÇÕES BÁSICAS

Ruby permite avaliar expressões aritméticas tradicionais:





A screenshot of a terminal window titled 'ruby'. The window has three colored window control buttons (red, yellow, green) in the top-left corner. The terminal shows an IRB session with the following text:

```
$> irb
irb(main):001:0> 3 * (2 + 5) / 8
=> 2
irb(main):002:0>
```

Podemos verificar isso usando `.class`

Números inteiros muito grandes são convertidos automaticamente para Bignum, de forma a não sofrerem perda de precisão

Qualquer número com casas decimais é do tipo Float

Quando trabalhamos com números, provavelmente faremos operações matemáticas também. Para isso, devemos usar os operadores apropriados. Os principais operadores aritméticos em Ruby são:

- ```
irb(main):001:0> 2 + 2
=> 4
irb(main):002:0> 5 - 3
=> 2
irb(main):003:0> 10 / 2
```

```
=> 5
irb(main):004:0> 15 * 2
=> 30
irb(main):005:0> 3 ** 2
=> 9
```

Agora temos um problema: Se tentarmos fazer uma divisão cujo resultado não seja um número inteiro, este será arredondado para baixo!

```
irb(main):001:0> 15/2
=> 7
```

Para resolver esse problema, podemos usar um Float no lugar de Fixnum, em qualquer lado da operação:

```
irb(main):002:0> 15.0 / 2
=> 7.5
```

## Ranges

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.

## Símbolos

**Símbolos** também são texto, como as Strings. Só que devem ser precedidos do carácter ': ', ao invés de aspas e pertencem à classe Symbol:

```
>> puts :simbolo
simbolo
=> nil
>> :simbolo.class
=> Symbol
```

As principais diferenças são:

- São imutáveis. Uma vez criado, um símbolo não pode ser alterado. Se precisarmos de um novo valor, precisa criar um novo objeto símbolo.
- São compartilhados. Símbolos com o mesmo valor, mesmo que em locais diferentes do código, compartilham do mesmo objeto. Isto é, qualquer lugar da aplicação que contenha, por exemplo, :writable, se referem a um mesmo objeto do tipo Symbol.

Por causa destas características, símbolos são análogos às Strings do Java. As

Strings do Ruby estão mais para o `StringBuilder` do Java. Por serem imutáveis e compartilhados, objetos `Symbol` geralmente são usados como identificadores e para nomenclatura (labels). Durante o curso usaremos muito este recurso.

### 3.10 - EXERCÍCIOS - TIPOS

1. Em Ruby tudo é objeto. Vamos fazer alguns testes curiosos e descobrir os tipos:

```
# Tipos
puts 3.class
puts 33333333333333333333333333333333.class
```

2. Podemos fazer as operações básicas em ruby de forma simples:

```
# imprime conta
puts 3*(2+5)/8
```

### 3.11 - ESTRUTURAS DE CONTROLE

Para utilizar estruturas de controle em ruby, precisamos antes conhecer os operadores booleanos, `true` e `false`. Os operadores booleanos aceitam quaisquer expressões aritméticas:

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

Os operadores booleanos são: `==`, `>`, `<`, `>=` e `<=`. Expressões booleanas podem ainda ser combinadas com os operadores `&&` (and) e `||` (or).

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`:

```
>> variavel = nil
=> nil
>> if(variavel)
>>   puts("so iria imprimir se variavel != null")
>> end
=> nil
>> if(3 == 3)
>>   puts("3 é 3")
>> end
3 é 3
```

```
=> nil
```

Teste 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:

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

Ruby possui bom suporte a expressões regulares, fortemente influenciado pelo Perl. Expressões regulares literais são delimitadas por / (barra).

```
>> /rio/ =~ "são paulo"
=> nil
>> /paulo/ =~ "são paulo"
=> 4
```

O operador =~ faz a função de match e retorna a posição da String onde o padrão foi encontrado, ou nil caso a String não bata com a expressão regular. Expressões regulares aparecem com uma frequência maior em linguagens dinâmicas, e, também por sua sintaxe facilitada no Ruby, utilizaremos bastante.

## Rubular.com

Um site muito interessante para testar Expressões Regulares é o <http://rubular.com/>

## MatchData

Há também o método `match`, que retorna um objeto do tipo `MatchData`, ao invés da posição do match. O objeto retornado pelo método `match` contém diversas informações úteis sobre o resultado da expressão regular, como o valor de agrupamentos (captures) e posições (offset) em que a expressão regular bateu.

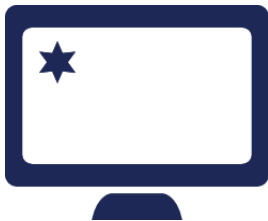
## Operador ou igual

O operador `||=` atribui um valor apenas a variável esteja vazia. é muito utilizado para carregar valores de maneira "lazy".

```
nome ||= "anonimo"
```

Nesse caso, se `nome` é nulo, ele será preenchido com "anonimo".

## Agora é a melhor hora de aprender algo novo



Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos novos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

[Conheça a Alura.](#)

## 3.12 – EXERCÍCIOS – ESTRUTURAS DE CONTROLE E REGEXP

1. Nosso restaurante pode ter notas boas se a mesma for superior a 7, ou ruim caso contrário.

```
# estruturas de controle: if
nota = 10

if nota > 7
  puts "Boa nota!"
else
  puts "Nota ruim!"
end
```

Teste com notas diferentes e verifique as saídas no terminal.

2. Implemente um for em ruby:

```
# estruturas de controle: for
for i in (1..3)
  x = i
end
puts x
```

3. Teste o operador de expressões regulares ""

```
puts /rio/ =~ "são paulo" # nil
puts /paulo/ =~ "são paulo" # 4
```

Abra o site <http://rubular.com/> e teste outras expressões regulares!

4. O operador "||=" é considerado uma forma elegante de fazer um if. Verifique se uma variável já foi preenchida, caso não estiver, vamos atribuir um valor a ela.

```
restaurante ||= "Fogo de Chao"
puts restaurante
```

### 3.13 – DESAFIOS

1. Sem tentar executar o código abaixo, responda: Ele funciona? Por que?

```
resultado = 10 + 4
texto = "O valor é " + resultado
puts(texto)
```

2. E o código abaixo, deveria funcionar? Por que?

```
puts(1+2)
```

3. Baseado na sua resposta da primeira questão, por que o código abaixo funciona?

```
resultado = 10 + 3
texto = "O valor é: #{resultado}"
```

4. Qual a saída deste código?

```
resultado = 10 ** 2
puts('o resultado é: #{resultado}')
```

5. **(Para Casa)** Pesquise sobre outras maneiras de criar Strings literais em Ruby.

6. Por que a comparação entre símbolos é muito mais rápida que entre Strings?

```
simbolo1 = :abc
simbolo2 = :abc
simbolo1 == simbolo2
# => true
```

```
texto1 = "abc"
texto2 = "abc"
texto1 == texto2
# => true
```

CAPÍTULO ANTERIOR:

[A linguagem Ruby](#)

PRÓXIMO CAPÍTULO:

[Mais Ruby: classes, objetos e métodos](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter