SOBRE O TUTORIAL

Este tutorial cobre alguns aspectos da linguagem Ruby, mostra de forma simples algumas de suas características através de exemplos e curtas descrições de conceitos, não se propondo ser um guia completo ou bíblia(:)), mas sim um guia rápido onde você poderá conhecer algumas dessas características.

Ele é direcionado a programadores que já possuam experiência e conhecimento em programação estruturada, orientada a objetos e desenvolvimento baseado em camadas.

Os assuntos abordados foram adicionados de acordo com o estudo que realizei durante o aprendizado dessa linguagem.

Djalma C. Oliveira F.

SOBRE A LINGUAGEM RUBY

Ruby foi criado por Yukihiro Matsumoto's (Matz) no Japão, lançando sua primeira versão em 1995, decisão essa baseada em uma busca por uma linguagem mais poderosa que Perl e mais orientada a objetos que Python após anos programando orientado a objetos.

A filosofia de Ruby é baseada no Principle of Least Surprise , significa que a linguagem funcionará da forma que você espera que ela funcione.

Possui uma sintaxe simples, suporta orientação a objetos, é dinâmica entre outras características.

Ela foi desenvolvida utilizando as melhores características de algumas outras linguagens de programação, tornando-a muito flexível e poderosa.

Em Ruby tudo é objeto(referência a objetos), como em algumas outras linguagens.

Sua licença de uso é a GPL e possui versões para os principais sistemas operacionais como GNU/Linux, MacOS, DOS, Win32, alguns Unix entre outros.

Com a sua tipagem dinâmica, suas variáveis são automaticamente tipadas através do seu conteúdo no decorrer da aplicação.

VARIÁVEIS

Nomenclatura de variáveis

Todas os nomes de variáveis devem iniciar com letra minúsculas.

A primeira letra deve ser um caracter alfabético ou _ , não sendo permitido nenhum número.

Os tipos são atribuídos automaticamente na primeira inicialização de acordo com o valor atribuído, ou nas reatribuições através do código.

Para descobrir a classe atual de qual a variável foi classificada usa-se o método ***.class*** da variável.

Ex.:

```
nome = 'Silva'
idade = 30
_sexo = 'M'
puts _sexo.class # String
puts idade.class # Fixnum
```

Variáveis Locais

Variáveis locais são declaradas simplesmente assim:

Ex.:

```
def teste
      x = 12
end
```

A visibilidade da variável *x* é utilizada somente dentro do método *teste*, ou seja, você somente poderá manipular essa variável dentro desta função.

Variáveis Globais

Variáveis globais, podem ser acessadas em qualquer lugar. Elas são identificadas usando o caracter $ no inicio de seu nome.

Ex.:

```
$projeto = 'ruby'   # $projeto é uma variável global
def meuprojeto
      projeto = 'outro projeto'  # projeto é uma variável local da função
      puts $projeto
end
meuprojeto
```

Tipos

Os tipos de variáveis padrão do Ruby são: numbers, strings, ranges e regular expression.

Ex.:

```
Numbers
      num = 8       # Fixnum
```

```
        10              # Fixnum

Strings
        "maria"
        nome = "maria"

Ranges
        numeros  = 1..10
        alfabeto = 'a'..'z'

Regular Expression
        b = /^maria/
```

STRINGS

São praticamente textos declarados entre aspas. Mesmo sendo simples trechos textuais também são considerados objetos, possuindo métodos que o manipulam.

Ex.:

```
a = "esse é uma string"
b = "aqui é outra string"
c = 'esse também é uma string'
```

Quando são declarados entre aspas são chamadas strings literais.

Aspas podem representar textos somente em uma linha, para representar textos em múltiplas linhas você deve utilizar um *quotation mark* representado por %q{ e } ou %Q{ e }. Os delimitadores {} podem ser qualquer outro como (), [], !!, <> etc.

%q representam strings escapados com aspas simples.

%Q representam strings escapados com aspas duplas.

Ex.:

```
a = %q{
      varias
      linhas de texto
}
nome = 'maria'
b = %Q{ seu nome é: #{nome} }
puts a
puts b
```

Outra forma de criar strings é utilizando *here document.*

Ex.:

```
a = <<FIMDASTRING
      mais um texto
      de varias
      linhas
FIMDASTRING
puts a
```

Para adicionar variáveis dentro de uma string literal a mesma deve ser escapada entre aspas duplas ou %Q ou usada no *here document.*

Utilizamos a construção *#{variavel}* dentro da string.

Ex.:

```
nome = 'maria'
b = %Q{ seu nome é: #{nome} }
puts b
```

ARRAY

Array é uma coleção de objetos, é usado para armazenar um conjunto de objetos sejam, números, textos, objetos instanciados entre outros. O acesso por índice dos Arrays tem inicio no 0.

Sintaxe:

variável = [val1, val2, val3...]

Ex.:

```
nomes = ['maria', 'joao', 'marcio']
puts nomes.class
puts nomes[0]# primeiro valor array
x = []       # array vazio
```

Adicionando valores no Array

Uma das forma de inserir valores no Array é utilizando o operador <<.

Ex.:

```
a = ['maria']
a << 'joao'
a << 'marcio'
puts a
```

Existem alguns métodos para manipulação de Arrays, você pode conferí-los dando uma olhada no método *.methods* do objeto.

HASHES

Como os Arrays, Hashes são coleções de objetos, mas a forma como é armazenado os valores é diferenciado. Hashes não são adequados para armazenar dados que devem ser ordenados, Arrays nesse caso seria mais indicado.

A sua construção

Ex.:

```
maria = {'nome'=>'maria', 'idade'=>'25'}
puts maria['nome']
puts maria['idade']
maria.each do |i, v|
	puts i
	puts v
end
```

SYMBOLS

Um *symbol* é um objeto instanciado da classe Symbol. Sua construção é realizada utilizando o caracter : no início do identificador.

Seu uso é muito semelhante a strings, mas nem tudo que pode ser feito com strings pode ser usado com *symbol*. A grande vantagem de usá-lo é que uma vez criado um *symbol* ele sempre existirá e será o mesmo durante todo funcionamento da aplicação.

Symbols não recebem valores, o nome dele já representa o seu valor em string sem o :.

Ex.:

```
puts   :nome
a =    :ab
puts a
puts "teste #{:novo}"
maria = {:nome=>'maria', :idade=>'32'}
maria.each do |i,v|
       puts i
       puts v
end
```

ALIAS

Cria um novo nome para métodos, operadores, variáveis globais ou expressão regular.

Quando é realizado um *alias* de métodos o *novonome* conterá uma nova cópia daquele método, mantendo assim seu bloco de instruções mesmo que o método de origem seja modificado posteriormente.

Sintaxe:

alias novonome nomeorigem

Ex.:

```
$nome = 'maria'
puts $nome.object_id
alias $anome $nome
puts $anome.object_id

def original
      puts 1+2
end
original
alias copia original
def original
      puts 2+2
end
original
copia
```

CONTROLE DE FLUXOS

if

Executa instruções se a condição retornar verdadeira e executa outras instruções se retornar falsa.

Sintaxe:

```
if <condição>
        [instruções se verdadeiro]
else
        [instruções se falso]
end

if <condição>
        [instruções se verdadeiro]
end
```

Ex.:

```
idade = 17
if idade > = 18
       puts "Você pode dirigir."
else
       puts "Você não está pronto para dirigir."
end
```

unless

Seria o mesmo que dizer: Não sendo <condição> .

Sintaxe:

```
unless <condição>
        [instruções]
end
```

Ex.:

```
idade = 17
unless idade >= 18
       puts "Você não tem idade para dirigir."
end
```

Operador ternário ?:

Retorna um dos valores declarados dependendo do resultado da condição.

Sintaxe:

```
variável = <condição> ? valor_se_verdadeiro : valor_se_falso
```

Ex.:

```
idade = 17
msg = idade >= 18 ? "Pode dirigir." : "Não pode dirigir."
puts msg
```

elsif

É o mesmo que *if <condição>*, mas aninhado a estrutura do if.

Sintaxe:

```
if <condição>
```

[instruções]
elsif <outracondição>
[instruções]
elseif <maisoutracondição>
[instruções]
end

Ex.:

```
nota = 5
if nota <= 5
	puts "Nota ruim."
elsif (nota > 5) && (nota <= 8)
	puts "Nota boa."
elsif (nota > 8)
	puts "Nota ótima."
end
```

case

Executa um bloco de instruções de acordo com o valor da variável de controle.

case também pode retornar valores da última instrução executada no bloco, sendo só necessário fornecer o resultado a uma variável.

Sintaxe:

variável = case <variávelcontrole>
or
case <variávelcontrole>
when valor1
[instruções se valor1]
when valor2
[instruções de valor2]
else
[instruções senão casar nenhum valor]
end

Ex.:

```
nome = 'Maria'
case nome
	when "João"
		puts "Seu nome é João."
	when "Maria"
		puts "Seu nome é Maria."
	else
		puts "Seu nome não foi encontrado."
end
```

REPETIÇÕES(Loops)

while

Executa o bloco de instruções enquanto a condição retornar verdadeira, para quando for falsa.

Sintaxe:

```
while <condição>
    [instruções]
end
```

Ex.:

```
x = 1
while x <= 10
      puts x
      x = x + 1
end
```

until

Executa o bloco de instruções enquanto a condição retornar falsa, para quando for verdadeira.

Sintaxe:

```
until <condição>
    [instruções]
end
```

Ex.:

```
x = 1
until x == 11
      puts x
      x = x + 1
end
```

for

Executa instruções quantas vezes forem o valor da *expressão*, veja que *expressão* pode ser representado de várias formas, Array, Ranges entre outros.

As variáveis de interação contém os valores lidos da expressão em cada repetição, permanecendo a sua existência mesmo após a finalização da repetição.

Sintaxe:

```
for variavelinteração,[variavelinteração]    in expressão
    [instruções]
end
```

Ex.:

```
for i in 1..10
      puts i
end
puts i
nome = ['maria','joao']
for i in nome
      puts i
end
puts i
```

each

Efetua iteração em cada item do objeto. e*ach* é um método do objeto não uma construção da linguagem.

Sintaxe:

expressao.*each do |p1,p2|*

[instruções]

end

Ex.:

```
nomes = ['maria', 'joao']
nomes.each do |nome|
      puts nome
end
```

loop

Efetua repetições até encontrar uma instrução de parada *break*.

Sintaxe:

loop do

[instruções]

end

Ex.:

```
x = 10
loop do
      puts x
      if x == 0
            break
      else
            x = x -1
      end
end
```

Controlando o fluxo das repetições

break [valor]

Interrompe o fluxo de execução do bloco corrente.

Se especificado um valor sai do bloco retornando o valor especificado.

redo

Reinicia a execução do bloco de repetição desde o início, sem processar a condição ou incrementar valores em uma interação.

next [valor]

Pula o restante do bloco e efetua mais uma repetição normalmente, avaliando a condição e incrementando valores. (seria o mesmo que o *continue* em algumas outras linguagens de programação)

Se especificado um valor e está num loop o valor é ignorado.

retry

Semelhante ao redo, reinicia a repetição mas avalia a condição.

MÉTODOS

São blocos de código que executam um conjunto de instruções. Não confunda com métodos de objetos, neste caso estamos falando de métodos avulsos , semelhante a funções comparando a algumas outras linguagens de programação.

A declaração é realizada com a palavra-chave ***def*** e finalizado com ***end***.

Ex.:

```
def minhafuncao
       puts 'executando a minha função'
end
minhafuncao()       # duas formas de chamada para o método
minhafuncao
```

Retornando Valores

Uma função retorna valores quando a última expressão na função é avaliada, ou utilizando a instrução ***return.***

Ex.:

```
def meunome
       'Natalia'
end
def minhaidade
       return 17
end
puts meunome
puts minhaidade
```

Passagem de valores

Pode ser realizado através da declaração de argumentos na definição do método.

Ex.:

```
def somar(n1, n2)   # n1 e n2 são argumentos que receberão valores
       n1 + n2
end
puts somar(1,2)     # escreve 3
puts somar 1,3      # outra forma de chamar a função, escreve 4
```

Os argumentos podem vir com **valores predefinidos** veja:

```
def somar(n1, n2=10)
       n1+n2
end
puts somar(1)       # 11
puts somar(1,11)    # 12
```

Também podemos passar um **número variável de argumentos**, usamos o caracter * na frente do nome do argumento, veja:

```
def somar(*numeros)
       puts numeros.class  # Array
       total = 0
       for i in numeros
```

```
            total += i
        end
        total
end
puts somar(3,5)
```

Como você pode perceber o argumento ***numeros*** é um Array, com os valores de todos os argumentos passados.

Blocos de instruções

Bloco de instrução é um conjunto de instruções que são declarados entre chaves *{}* ou *do end.* Veja um bloco como um método ou função sem nome, anônima, mas que possui instruções a serem executadas.

O bloco pode conter também uma lista de argumentos, eles são declaradas entre | e |, separados por vírgulas.

Igualmente como os métodos, os blocos seguem a mesma teoria de escopo de variáveis e outras estruturas que forem declaradas dentro da mesma.

Ex.:

```
{ |arg1, arg2|
	puts "oi"
}

do |arg1, arg2|
	puts "oi"
end
```

Passando blocos de instrução para execução

Existem diversas formas de passar blocos de instruções para serem executados.

Uma delas é usar *yield* dentro da declaração de um método para indicar que o bloco passado ao método deve ser executado, veja *yield* como o próprio bloco passado como parâmetro.

Ex.:

```
def meunome
	puts "vou escrever o meu nome"
	yield
	puts "agora terminei de escrever"
end
meunome {puts "meu nome é João"}
```

Se quiser passar parâmetros para o bloco, você deve declarar *yield* com parâmetros e declarar os argumentos no bloco.

Ex.:

```
def somar
	yield(1,6)
end
somar {|n1, n2|
		puts n1+n2
	}
```

Blocos também podem ser usados para realizar interações com objetos.

Ex.:

```
$pessoas = ["Maria", "José", "Joaquim"] # array
$pessoas.each { |nome| puts nome }
# seria o mesmo que
def shownome
	for nome in $pessoas
		yield(nome)
	end
end
shownome {|nome| puts nome}
```

Blocos não são objetos, eles são somente blocos de instruções, mas podemos transformá-los em objetos utilizando a classe Proc.

A primeira forma é utilizando um argumento na definição de método utilizando o caracter & no início do nome.

Ex.:

```
def testaBloco(id, &bloco)
      puts id
      puts bloco.class    # Proc
      bloco.call          # chama a execução do bloco
end
testaBloco(10) {puts "eu sou um bloco"}
```

A segunda forma é criando um objeto tipo Proc e atribuindo a uma variável.

Ex.:

```
bloco = Proc.new {puts "opa sou um bloco"}
puts bloco.class    # Proc
bloco.call          # executa o bloco
```

A terceira forma é utilizando o método Kernel.lambda.

Ex.:

```
bloco = lambda {puts "opa sou um bloco"}
puts bloco.class    # Proc
bloco.call          # executa o bloco
```

Dentro do bloco uma chamada de *next valor* faz com que o bloco retorne o *valor* passado. Se não for encontrado *next* retornará *nil*.

Ex.:

```
bloco = Proc.new {next "estou saindo"}
puts bloco.class    # Proc
puts bloco.call     # executa o bloco
```

E se no bloco uma chamada de *break valor* ou *return valor* for realizada, o método que chamou o bloco é parado naquele ponto e retornado o *valor* especificado em *break/return* do bloco.

Ex.:

```
def testaBloco(id, &bloco)
      puts id
      bloco.call              # chama a execução do bloco
      puts "não deve sair na tela"
end
puts testaBloco(10) {break "tchau"}
```

Tratando Exceções

Exceções são eventos que podem ocorrer no decorrer da execução de seu programa e podem gerar erros não esperados, por exemplo tentar gravar dados em um disco cheio.

Para capturar a mensagem que gerou a exceção você pode usar a variável especial *$!* .

Existem 3 formas de declaração do bloco que manipula exceções:

```
1°     # uso geral
begin
       instruções ...
[ rescue [ parm, ... ] [ => var ] [ then ]
       [instruções para manipulação de erros... , ... ]
[ else
       [instrução se não houver exceção... ]
[ ensure
       [sempre executa essas instruções... ]
end

2°     # dentro de métodos
def  nome_metodo
       instruções ...
       [ rescue [ parm, ... ] [ => var ] [ then ]
              [instruções para manipulação de erros... , ... ]
       [ else
              [instrução se não houver exceção... ]
       [ ensure
              [sempre executa essas instruções... ]
end

3°     # junto com a expressão na mesma linha
instrução    [ rescue instruções ]
```

Digamos que você queira efetuar uma conexão com um servidor ftp, mas a aplicação Ruby não sabe se o servidor estará disponível quanto realizar a conexão, dessa forma tratamos estas exceções usando o bloco de tratamento como nos exemplos abaixo:

```
1°
require 'net/ftp'
begin
       ftp = Net::FTP.new('ftp.site.org')
rescue
       puts "Não foi possível efetuar a conexão: "+$!
end

2°
def conectaftp
       ftp = Net::FTP.new('ftp.site.org')
       rescue
              puts "Não foi possível efetuar a conexão: "+$!
end
conectaftp

3°
Net::FTP.new('ftp.site.org') rescue puts "Não foi possível efetuar a conexão: "+$!
```

Forçando uma exceção

Em algumas situações você gostaria de disparar suas próprias exceções, isso é possível utilizando o

raise.

Existem 3 formas de uso:

```
1°    raise
2°    raise string
3°    raise thing[, string [stack trace]]
```

CLASSES E OBJETOS

Objetos

Tudo em Ruby é um objeto, uma string, um número, um método por exemplo.

Todos os objetos são herdados do objeto principal da linguagem que chama-se *Object*.

As variáveis não são objetos, são referências para um objeto, você pode ter várias variáveis apontando para um único objeto por exemplo.

Sendo objetos eles possuem métodos e atributos pré-definidos.

Ex.:

```
"maria".class       # String
2.class             # Fixnum
x = "marcos"
x.upcase            # MARCOS
```

Para ver quais métodos foram definidos para o objeto utilize o método *.methods*, *veja*:

```
puts x.methods      # retorna a lista de métodos atribuídos atualmente ao objeto
```

E para inspecionar como o objeto foi definido usa-se *.inspect*, veja:

```
puts x.inspect      # mostra "marcos" exatamente como foi definido o seu valor
```

Classes

Classes são representações genéricas(estrutura) de algum objeto. Um objeto pode ser qualquer coisa real ou imaginária, como: pessoa, processo de venda, carro, processo de transação bancária, conta corrente, cliente, fornecedor, boneca entre outros.

Essa representação se dá em duas formas:

- atributos(características dos objetos) ;
- métodos(as ações que o objeto pode possuir);

Se considerarmos um objeto ***pessoa***, que características e ações ela poderia possuir?

Características: nome, sexo, idade, endereço etc.

Ações: andar, correr, pular, parar, falar etc.

As classes são usadas para representar uma parte de um todo que você deseja idealizar.

Em Ruby definimos uma classe usando a palavra-chave ***class*** e finalizamos com ***end***.

Ex.:

```
class Pessoa

end
```

Variáveis de Instância (Objetos)

São os atributos do objeto, elas estão disponíveis apenas após a instancialização do objetos.
Essas variáveis são identificadas com caracter @ no inicio de seu nome.
Ex.:

```
@nome
@idade
@sexo = 'M'         # inicialização
```

Essas variáveis normalmente são declaradas dentro de um método especial chamado *initialize*, mas podem ser declaradas em qualquer outro método da classe.

Mas por definição da linguagem essas variáveis são *private*, ou seja, não podem ser acessadas externamente, para que seja possível o seu acesso se faz necessário a criação de métodos tipo 'get' para retornar os valores dessas variáveis, bem como também podem ser criados métodos tipo 'set' para a atribuição de novos valores para elas.

Ex.:

```
class Minhaclasse
       def initialize
              @nome = 'maria'
       end
end
pessoa = Minhaclasse.new
puts pessoa.nome    # causaria um erro nesta linha
```

Então para que se possa acessar a variável ***@nome*** deve-se declarar o seguinte método dentro da classe:

```
class Minhaclasse
       def initialize
              @nome = 'maria'
       end

       # metodo tipo 'get' para acessar a variável de instância
       def nome
              @nome
       end

end
pessoa = Minhaclasse.new
puts pessoa.nome    # escreveria maria
```

Perceba aqui que a variável de instância não pode acessada diretamente fazendo-se necessário a definição de métodos de acesso para a mesma.

Métodos de Instancia (Objetos)

Métodos de objetos são basicamente as ações que um objeto pode possuir. Elas são declaradas dentro da definição de classe usando a palavra-chave ***def*** e terminadas com ***end***.

Ex.:

```
class Pessoa
	# declarção de um método
	def shownome
		puts 'Mario'
	end
end
p1 = Pessoa.new	# instancialização
p1.shownome
```

Existe um método de inicialização do objeto chamado de *initialize*. Esse é o primeiro método é executado todas as vezes que um objeto é instanciado (objeto.new), e possui a função de inicializar variáveis do objeto, além de outros itens que o objeto necessitar. Ele possui visibilidade private, ou seja, não pode ser acessado fora do objeto.

Como todo método, também pode-se utilizar passagem de valores, que podem pré-inicializar os atributos do objeto, por exemplo.

Ex.:

```
class Pessoa
	def initialize(nome,idade)
		@nome = nome
		@idade = idade
	end

	def nome	# 'getter'
		@nome
	end

	def idade	# 'getter'
		@idade
	end

	def nome=(nome)	# 'setter'
		@nome = nome
	end

	def idade=(idade)	# 'setter'
		@idade = idade
	end
end
maria = Pessoa.new(“maria”,19)
puts maria.nome
puts maria.idade
maria.nome = 'Márcia'
maria.idade = 30
puts maria.nome
puts maria.idade
```

Métodos 'setter' como visto no código acima atribuem valores as variáveis de instância *@nome* e *@idade*, perceba o '=' na escrita de sua declaração *def idade=(idade)* ... .

Variáveis de Classe

São variáveis declaradas para uso somente da classe. Elas são identificadas com caracter @@ no inicio de seu nome e podem ser declaradas dentro do método ***initialize***.

Essas variáveis são utilizadas para declarar unicamente variáveis em todos os objetos instanciados por ela, ou seja, se você tiver *n* objetos instanciados, esta variável existirá em todos esses objetos e conterá o mesmo valor, seja qual for o objeto que tente acessá-la ou modificá-la.

Veja ela como sendo uma variável ***static***, mas utilizado dentro da classe.

Ex.:

```
class Pessoa
	def initialize
		if defined?(@@quantidade)
			@@quantidade += 1		# inicia a contagem a cada instância
		else
			@@quantidade = 1		# definindo a variável se não existir
		 end
	end

	def quant
		@@quantidade
	end
end

a = Pessoa.new
puts a.quant		# 1
b = Pessoa.new
puts a.quant		# 2
puts b.quant		# 2
```

Métodos de Classe

Métodos de classe possuem basicamente a mesma funcionalidade dos métodos de objetos, mas com diferença de que não é necessário instanciar o objeto para que ela possa ser acessada.

O nome do método deve possuir a palavra chave ***self.***, ou ***Nomedaclasse.*** no inicio de seu nome.

Ex.:

```
class Homem
	def self.showsexo   # ou Homem.showsexo
		puts 'Masculino'
	end
end
Homem.showsexo
```

Perceba que não é necessário instanciar nenhum objeto para que o método ***showsexo*** fosse executado, entretanto ele não pode ser acessado através de algum objeto instanciado, somente através da classe.

Estes métodos são úteis quando queremos visualizar as variáveis de classe definidas, e entre outras informações úteis sobre a utilização de objetos instanciados a partir desta classe.

Ex.:

```
class Homem
	def initialize
		if defined?@@quantidade
			@@quantidade +=1
		else
			@@quantidade = 1
		end
	end

	def Homem.showquant
		puts @@quantidade
	end
end
a = Homem.new
Homem.showquant     # escreve 1
b = Homem.new
Homem.showquant     # escreve 2
```

Definindo variáveis de instancia (Objetos)

Para criar variáveis/atributos para os objetos declaravamos a variável de instancia(@), depois métodos que permitiam o seu acesso('getter') e sua atribuição('setter') como no exemplo:

```
class Pessoa
	def initialize
		@nome = ''	# declaração e inicialização das variáveis/atributos do objeto
	end
	def nome		# método 'getter'
		@nome
	end
	def nome=(nome)		# método 'setter'
		@nome = nome
	end
end
maria = Pessoa.new
maria.nome = 'maria'
puts maria.nome
```

Mas o Ruby possui alguns métodos que fornecem atalhos para essa criação, são eles: *attr_accessor*, *attr_reader* e *attr_writer*.

Sintaxe:

```
	attr_accessor :atributo1, :atributo2, ...
	attr_reader   :atributo1, :atributo2, ...
	attr_writer   :atributo1, :atributo2, ...
```

O *attr_accessor* cria automaticamente os métodos de 'getter' e 'setter' na classe junto com a variável de instancia de mesmo nome do *symbol*.

O *attr_reader* cria a variável de instancia e somente o método 'getter', ou seja torna-a somente para leitura.

E o *attr_writer* cria a variável de instancia e somente o método 'setter', ou seja, torna-a somente para escrita.

Ex.:

```
class Pessoa
	attr_accessor :nome, :idade
end
maria = Pessoa.new
puts maria.nome		# nil
maria.nome = 'maria'
puts maria.nome		# maria
```

Como pode ver, se você não precisa especificar nenhuma instrução para os métodos das classes que você desenvolver, então utilize este tipo de declaração, caso contrário, você pode definir manualmente cada método 'getter/setter' ou sobrepor os médotos.

Outra forma de criar instancias de objetos é utilizando a classe Struct.

Ex.:

```
Pessoa = Struct.new(:nome, :idade) # define os atributos da nova classe a constante Pessoa
maria = Pessoa.new('maria', 29)    # inicializa normalmente
puts maria.nome
```

Redefinindo funcionalidades das classes ou métodos (sobreposição)

Uma característica interessante do Ruby é ser uma linguagem dinâmica, ou seja, você pode a qualquer momento acrescentar ou modificar funcionalidades(métodos, variáveis, visibilidade etc), as suas classes ou métodos mesmo após sua declaração. É necessário somente declarar novamente a classe ou método e acrescentar os novos métodos a ela.

Ex.:

```
class Pessoa
	def initialize(nome)
		@nome = nome
	end
end
maria = Pessoa.new('maria')
class Pessoa
	def  nome
		@nome
	end
end
puts maria.nome
```

Uma observação interessante sobre a sobreposição, é que ela pode ser realizada também com as classes já definidas da linguagem Ruby, como por exemplo a classe Fixnum.

Adicionando novas funcionalidades a objetos instanciados

Você pode adicionar novos métodos, por exemplo, a objetos já instanciados durante a execução de sua aplicação.

Umas das formas de realizar é utilizando o operador <<*obj* , onde *obj* é o objeto já instanciado.

Ex.:

```
pessoa = "maria"    # é uma simples variável, mas em Ruby tudo é objeto
class <<pessoa
	def shownome
		puts self
	end
end
pessoa.shownome     # maria
```

O que o código acima faz é : adicione ao objeto pessoa o método *shownome*.

Visibilidade dos métodos

Existem 3 formas de visibilidade dos métodos:

- public
- private
- protected

O *public* não precisa de uma declaração explícita, pois as próprias declarações de métodos dentro da classe são por definição públicas, ou seja, podem ser acessadas fora do objeto.

O *private* deve ser declarado explicitamente, são métodos privados, ou seja, só podem ser acessados dentro do objeto.

O *protected* também deve ser declarado explicitamente, são métodos protegidos, ou seja, só podem ser acessados dentro do objeto ou por objetos descendentes dele(filho).

Exite duas formas de efetuar a declaração:

```
# 1º
class Pessoa
	public		# não é necessário, por definição elas já são públicas
		def nome
			puts "meu nome"
		end
	private
		def clona_nome
			puts "clonando"
		end
	protected
		def metodo_protegido
			puts "estou protegido"
		end
end

# 2º
class Pessoa
	def nome
		puts "meu nome"
	end
	def clona_nome
		puts "clonando"
	end
	def metodo_protegido
		puts "estou protegido"
	end

	public		:nome			# semelhante a symbol
	private		:clona_nome
	protected	:metodo_protegido
end
```

Uma observação para a segunda forma, é que os métodos devem ser todos declarados antes de serem atribuídos a sua visibilidade.

Herança

Para realizar herança utilizamos a expressão < *classe_pai* após o nome da classe.

Instanciação de subclasses não inicializa automaticamente o método *initialize* da classe pai, sendo necessária chamá-la explicitamente no método *initialize* da subclasse, para que isso ocorra é necessário que seja chamado a instrução *super*, já os métodos são herdados automaticamente.

O *super,* executa o método de mesmo nome no qual ele foi declarado na classe pai.

Ex.:

```
class Pessoa
	def initialize(nome)
		@nome = nome
	end
	def nome
		@nome
	end
end
class Homem < Pessoa	# herança
	def initialize
		super		# é o mesmo que chamar o método initialize da classe pai (Pessoa)
		@sexo = 'Masculino'
	end
	def nome
		super		# é o mesmo que chamar o método nome da classe pai (Pessoa)
	end
end

jose = Homem.new
puts jose.inspect
```

E se você quiser executar diretamente o método *nome* definido na classe Pessoa(pai) através da instância(objeto) Homem?

INCLUSÃO DE ARQUIVOS

Existem 2 formas de incluir arquivos em Ruby, usamos *load ou require*.

load 'arquivo.rb'

Inclui em sua aplicação o código do *arquivo.rb* cada vez que *load* é chamado, muito usado para incluir arquivos que são modificados constantemente.

require 'arquivo'

Inclui em sua aplicação o código do *arquivo* uma única vez, não requerendo chamadas sucessivas. *require* também pode ser usado para carregar bibliotecas compartilhadas.

O caminho para os arquivos incluídos podem ser fornecidos através do caminho relativo ou absoluto (caminho completo desde a raiz).

O path de procura padrão no Ruby é representado por um Array *$:*, se precisar adicionar algum novo local é só adicioná-lo a essa variável predefinida global.

Variáveis locais declaradas nos arquivos incluídos não são repassadas para sua aplicação, somente as outras estruturas.

MÓDULOS

Módulos são agrupamentos de instruções(classes, métodos etc) que representam uma unidade de apoio ao desenvolvedor, veja ele como bibliotecas de novas funcionalidades que podem ser inseridas na sua aplicação.

Os módulos fornecem meios de diferenciar as suas implementações das já existentes na sua aplicação ou na linguagem Ruby, chamamos essa característica de *namespaces*.

Ex.:

```
# arquivo meumodulo.rb
module MeuModulo
	def MeuModulo.showme
		puts "oi sou um módulo"
	end
end

# arquivo teste.rb – o qual você deve executar
require 'meumodulo'
MeuModulo.showme
```

Existe outra forma de adicionar métodos de classe a partir do módulo, para isso declaramos os métodos do módulo sem o nome do módulo(como um método comum) e chamamos dentro da declaração de classe o método *extend nomedomodulo*.

Ex.:

```
module Teste
	def showme
		puts 'módulo teste'
	end
end
class Pessoa
	extend Teste
end
Pessoa.showme
```

Você deve atentar para alguns detalhes na criação dos módulos:

- palavra-chave *module* em minúsculo;
- o nome do módulo deve iniciar com letra maiúscula *MeuModulo*, igual as constantes e classes*;*
- os métodos declarados no módulo são realizados da mesma forma que os *métodos de classe*;

Mixins

Mixin é uma meio de incluirmos métodos declarados em módulos ficar disponível como métodos de instancias nas classes. Usamos a expressão *include NomedoModulo* para isso.

Ex.:

```
# arquivo meumodulo.rb
module MeuModulo
	def showme
		puts "oi sou um módulo"
	end
end

# arquivo teste.rb – o qual você deve executar
require 'meumodulo'
class Pessoa
```

```
      include MeuModulo          # inclusão do módulo
end
maria = Pessoa.new
maria.showme
```

Uma outra forma de mixin é utilizando o método *extend* do objeto instanciado.

Ex.:

```
novo_obj = "qualquercoisa"
novo_obj.extend MeuModulo
novo_obj.showme
```

Atente para a declaração do módulo quando você quer usá-lo como *mixin* :

- os métodos são declarados como *métodos de instância*.

A ligação *mixin* do módulo incluído na classe é realizada através de referências ao módulo, ou seja, não existe uma cópia para cada classe, sendo assim uma mudança no módulo faz com que todas as classes que utilizaam *mixin* com esse módulo refletira automaticamente essa mudança.

Uma observação final é que você pode declarar os métodos do módulo usando *métodos de instância* e *métodos de classe* ao mesmo tempo, mas o uso desses métodos vai depender de como esse módulo é usado.

Se esse mesmo módulo for chamado por N*omedomodulo.método* o método executado tem que ser de classe, se for incluído dentro de uma classe a chamada *objeto.metodo* tem que ser de instância.

BIBLIOGRAFIA

Sites

http://www.ruby-lang.org/en/

Livros

Peter Cooper, Beginning Ruby: From Novice to Professional

Dave Thomas, Programming Ruby - The Pragmatic Programmers Guide