Rápida Introdução a Ruby

Gubi

7 de março de 2019

Sumário

1 Introdução

Por que mais uma linguagem? Ou melhor, por que existem tantas linguagens de programação?

Todas as linguagens podem representar qualquer programa, afinal todas implementam uma máquina de Turing de uma forma ou de outra. O que realmente muda é a expressividade e a presença de construções que permitem trabalhar mais facilmente com um ou outro paradigma.

Assembly é a mais básica, praticamente a própria linguagem de máquina, e permite controle refinado sobre o código final e a execução. Em princípio é imbatível em termos de tamanho reduzido de código e velocidade de execução. Na prática, no entanto, o código fonte tende a ser gigante e a possibilidade de erros é grande, além da enorme dificuldade de depuração.

Linguagens de nível mais alto passam a responsabilidade de criar um código final "enxuto" e correto para o compilador. Com isso o programador pode se concentrar nos algoritmos e estruturas de dados e produzir um sistema mais sofisticado com mais facilidade. Só isto já torna natural a existência de linguagens específicas para domínios diferentes.

Além desta justificativa, há uma outra componente talvez mais importante. A evolução dos sistemas e formas de programação levou a criação de diversos paradigmas, como orientação a objetos e programação funcional. Embora esta última seja bem antiga, tem ganhado muita força recentemente.

A expressividade das linguagens de roteiro (*script languages*) e a velocidade de desenvolvimento fizeram com que elas se tornassem muito populares. Há um custo, no entanto: por um lado, estas linguagens trazem consigo uma grande parte de soluções prontas, mas a flexibilidade exige rotinas complexas e estruturas de dados maiores. Isto é compensado pelo aumento da capacidade de memória e da velocidade dos computadores e pela facilidade de manutenção e documentação de *software* que estas linguagens permitem.

A escolha da linguagem ideal depende da finalidade do projeto, de gosto pessoal, ou de circunstâncias extras (*software* disponível, existência de código legado, o professor mandou, etc).

A finalidade do projeto impõe requisitos naturais:

- Qual o balanço entre eficiência e manutenção?
- Qual o paradigma de desenvolvimento?
- Há código legado?
- Qual a capacitação da equipe?
- Qual o prazo para o desenvolvimento?

1.1 Perl e Python

Duas das *script languages* mais populares são Perl e Python. Elas possuem muita coisa em comum, mas possuem diferenças filosóficas radicais.

Perl foi projetada para o desenvolvimento rápido de programas para manipulação de texto. Com o tempo foi ampliando o seu alcance para se tornar de uso geral. Em Perl a sintaxe é muito flexível, de modo que o programador não tenha que pensar muito em como escrever o código. Basta ter a ideia e sair escrevendo (ou quase). A contrapartida é que como as variáveis possuem caracteres pouco usuais para identificar sua natureza, um *script* em perl tende a assustar os incautos. Possui suporte para programação funcional e orientação a objetos, ainda que de uma forma estranha. O lema da linguagem é "há sempre mais de uma forma de se fazer alguma coisa".

Sua principal qualidade é a integração fácil com o sistema operacional e a facilidade em trabalhar com expressões regulares. Os programas ficam curtos e aparentemente crípticos para os não iniciados, o que faz com que os "pythonistas" torçam o nariz.

Já Python é uma linguagem "purista". É bastante exigente na clareza do código e na forma em que as coisas são implementadas. Possui suporte à programação funcional e é de fato orientada a objetos e à meta-programação. Embora não perca a flexibilidade, o lema é "existe, em geral, uma forma ideal de se fazer as coisas".

A grande característica é a capacidade para modularização de código e a orientação a objetos propriamente dita. Os programas são ligeiramente mais verborrágicos do que em Perl, o que faz com que os monges do Perl (*perl monks*) torçam o nariz.

1.2 Ruby

Ruby foi criada por Yukihiro Matsumoto (vulgo Matz) em 1993, no Japão. O objetivo da linguagem é produtividade e simplicidade. De certa forma, ela combina forças de várias outras linguagens, em particular Perl e Python. Matz escreveu que queria uma linguagem mais poderosa do que Perl e mais orientada a objetos do que Python. O alvo era uma linguagem natural, mas não necessariamente simples, "assim como a vida".

O resultado é uma linguagem com a facilidade de escrita próxima a Perl (depois de se acostumar) e com um rigor próximo a Python. Seu suporte a expressões regulares é excelente e é completamente orientada a objetos, além de ter suporte avançado a programação funcional.

Esta discussão não reflete a história real de Ruby, mas minha visão do resultado final. Desde a criação de Ruby, tanto Perl como Python evoluíram muito, mas alguns paralelos com ambas linguagens fazem com que tanto monges e pythonistas torçam o nariz...

Brincadeiras a parte, Ruby é uma linguagem fácil e divertida de programar, que se adapta muito bem à criação de sistemas web, principalmente se aliada ao arcabouço específico chamado *Rails*.

2 Conceitos Básicos

Meu objetivo aqui não é ensinar a programar, mas mostrar Ruby para quem já conhece outras linguagens. Explicarei as principais diferenças com pequenos trechos de programa e discutirei os conceitos envolvidos. Há muito material de excelente qualidade online, este texto serve como uma referência e uma rápida introdução.

No que segue, a resposta do interpretador Ruby (`irb`) será apresentada `neste formato`.

A primeira coisa que se deve ter em mente com Ruby é que ***tudo*** é um objeto. Veja o exemplo abaixo, que usa o método `even?` para saber se um número é par

```
1769.even?
=> saída
```

O número 1769 é um objeto da classe `Fixnum`, cujo método `even?` indica se o objeto é par ou ímpar.

A presença de um `?` no nome do método pode parecer estranha, mas é permitida em Ruby se estiver no fim. Como veremos, alguns métodos também possuem um `!` no final. A ideia é usar o `?` para indicar métodos que retornam um valor booleano e `!` para métodos que alteram o estado do objeto. Esta convenção torna o código literalmente mais legível.

Para saber qual a classe de um determinado objeto, use o método `class`. Todo objeto também deve ter uma identificação única, que pode ser consultada com `object_id`.

```
1729.class
=> Fixnum
1729.object_id
=> 3459
```

2.1 Tipos básicos

As variáveis em Ruby não são declaradas, seu tipo é determinado na atribuição de valores. O tipo é do valor, não da variável.

```
x = 4
=> 4
x.class
=> Fixnum
x = "oi"
=> "oi"
x.class
=> String
```

Começaremos com constantes, que não formam um tipo, nem são constantes (calma, já explico). Qualquer variável que começa com uma letra maiúscula é considerada constante, independente do valor associado. É possível mudar o valor da constante, mas Ruby manda um alerta se isso acontecer:

```
Resposta = 42
=> 42
Resposta = 1729
(irb):149: warning: already initialized constant Resposta
(irb):148: warning: previous definition of Resposta was here
=> 1729
```

2.1.1 Números

Números em Ruby seguem o padrão de outras linguagens. Os tipos básicos são inteiros e ponto flutuante, só não se esqueça que divisão de inteiros produz resultado inteiro. Existe ainda um terceiro tipo básico, que são inteiros muito grandes, pertencentes a classe `Bignum`, faça um teste.

As coisas ficam mais divertidas quando se usa os métodos especiais. Este exemplo dá uma ideia do que se pode fazer.

```
puts "Hey Shrek"
10.times {puts "Nós já chegamos?"}
```

Retornaremos a este exemplo mais tarde, mas dá para perceber tudo o que é estranho?

O `puts` na primeira linha deve ser claro: simplesmente imprime o argumento. Note que não existe necessidade de parênteses, uma vez que a situação é clara. Em Ruby, os comandos terminam com um final de linha ou com `;`, como em Python.

A segunda linha é mais interessante. O método `times` faz com que o bloco à direita seja repetido o número de vezes representado pelo inteiro que o chamou. Pode-se pensar que o bloco é um argumento

do `times`, embora não seja exatamente isso. O bloco é chamado (executado) por `times` diversas vezes. Esta é uma forma econômica e eficiente do `for`.

Números literais podem conter '_', este caractere é ignorado e serve para tornar números grandes mais legíveis.

2.1.2 Strings

Em Ruby *strings* são mutáveis, ao contrário de Python, e permitem interpolação, como em Perl.

Existem diversas formas para representar *strings*, dependendo do resultado desejado.

A mais familiar é a tradicional delimitação por aspas ("): `"esta é uma string"`. Neste caso, é possível representar caracteres como *newline* e tab podem ser representados da maneira tradicional (`\n` e `\t`, respectivamente), exatamente como em C.

Mas isso não é tudo! Com aspas duplas, também bode-se incluir trechos com `#{...}`. O conteúdo entre chaves será executado e incluído na *string*, na posição correspondente.

```
x = 42; puts "A resposta é #{x}"
A resposta é 42
=> nil
puts "A resposta também é #{3*14}"
A resposta também é 42
=> nil
```

As aspas simples (') indicam uma *string* literal, sem a substituição dos caracteres acima (`'\n'` tem dois caracteres, enquanto `"\n"` tem apenas um).

Uma terceira variação que vem do bash e do Perl, são as aspas invertidas (`), um tanto divertidas e perigosas. Neste caso, o conteúdo entre elas é passada para o *shell* e a *string* será o conteúdo da saída padrão dos comandos executados.

```
puts `echo exemplo bobo`
exemplo bobo
=> nil
```

Adicionalmente, existe o valor "imediato", bem útil para *strings* que ocupam várias linhas. Também tem origem no *shell* e é mais fácil explicar por exemplos:

```
longa = <<Fim
Esta é uma cadeia com
várias linhas
três, no caso
Fim
outra = <<~End
  Esta outra também.
  Mas neste caso, com a presença de ~,
     os espaços no começo são removidos.
End
```

O identificador que aparece logo depois do `<<` ou `<<~` serve de delimitador e é usado para indicar o final da *string*.[1] Também é possível colocar aspas simples ou invertidas ao redor do delimitador, a *string* será tratada de acordo.

[1] Ao invés de `<<` pode-se usar `<<-`, neste caso, o delimitador não precisa estar no início da linha.

Em todos os casos também é possível incluir valores formatados no estilo de Python, com `%`.

```
puts "Produto: %-10.10s\nPreço  : %7.2"%['Açúcar', 4.80]
```

Finalmente, as aspas simples, duplas e invertidas podem ser substituídas, respectivamente, pelas seguintes construções: `%q`[delim]...[delim], `%Q`[delim]...[delim], `%x`[delim]...[delim]. [delim] pode ser qualquer pontuação ou parênteses, chaves ou colchetes balanceados.

2.1.3 Símbolos

Símbolos são nomes e funcionam como *strings* constantes, muito úteis para chaves de dicionários e identificadores. Os símbolos seguem as regras de formação de identificadores, começando com ':'.

```
c1 = "cadeia"
c2 = "cadeia"
s1 = :cadeia
s2 = :cadeia
puts c1.object_id
puts c2.object_id
puts s1.object_id
puts s2.object_id
```

2.1.4 Arrays

Arrays são similares aos de Python e funcionam como esperado. Índices podem ser negativos, contando a partir do final.

Existem várias maneiras de instanciar um *array*. Veja os exemplos abaixo:

```
a = [1,4, "blabla", [3,4]]
b = Array.new(20) # 20 elementos nil
c = Array.new(3, 'bla') # ["bla", "bla", "bla"]
d = Array.new(4) {|x| x*x}  # [0, 1, 4, 9]
e = Array(3..5) # [3,4,5]
f = Array.[](1,0,4, "!!") #  [1,0,4,"!!"]
```

A grande novidade, emprestada de Perl, é a possibilidade de construir um *array* de *strings* diretamente a partir de uma lista de palavras, para simplificar a inicialização.

```
lista = %w[um dois três quatro] # ["um", "dois", "três", "quatro"]
```

2.1.5 Hashes

Hashes ou dicionários também funcionam como os equivalentes em Perl ou Python. Adicionalmente, possuem diversos métodos interessantes que facilitam muito sua manipulação. Veremos alguns nos exemplos mais adiante.

2.1.6 Faixas

Faixas ou *ranges* são intervalos de valores que podem ser usados como índices ou listas para *for*, de modo similar ao que acontece em Python.

Em Ruby, são indicados por `(inicio..fim)` ou `(inicio...fim)`. Neste último caso "fim" não faz parte da sequência.

```
a = Array(3..6) # [3,4,5,6]
b = Array(3...6) # [3,4,5]
c = Array('aa'..'cx') # adivinhe o que sai
```

2.2 Estruturas de controle

As estruturas de controle são as usuais, apenas com algumas diferenças de sintaxe. É mais rápido ilustrar com exemplos:

```
# condicionais
if a > 4
   puts "#{a} é grande"
end

# ou, equivalentemente, para expressões simples
puts "#{a} é grande" if a > 4

# outra forma
if a > 4 then puts "#{a} é grande" end

# unless é o if com a condição invertida
puts "#{a} é grande" unless  a <= 4

# for usando uma enumeração
for i in 1..4
    puts i
end

# case ou switch
case a
  when 3 then puts "quase lá"
  when 4 then puts "em cima"
  when 5..10 then puts "sobrou"
end
```

Os testes executados por `when` usam um operador especial (`===`) que normalmente é explicado de uma forma muito confusa. O que `===` faz é uma verificação de continência, a expressão `a === b` responde à pergunta "`b` pode ser um elemento de `a`?". No último exemplo acima, perguntamos se `a` é um dos valores da enumeração `5..10`.

Os comandos `while` e `until` são usados da mesma maneira e também podem ser colocados no final da expressão.

```
x = 3
puts x-= while x > 0
```

Os comandos abaixo servem para controlar os laços:

- `break` — sai do laço corrente
- `redo` — reinicia a iteração corrente
- `retry` — refaz todo o laço
- `next` — pula para a próxima iteração

3 Funções, Blocos, Procedimentos e Lambdas

As funções em Ruby não possuem muitas surpresas, são como em Python.

```
def quadrado(x)
    return x*x
end
```

Note que não é necessário o ":" no final da primeira linha.

Um *bloco* é um trecho de código que pode receber parâmetros. É delimitado por chaves ou `do` e `end`. Parece uma função, mas é bem diferente. Para começar, não tem nome e não pode ser salvo em uma variável para referência posterior. Pode-se pensar no bloco como uma estrutura sintática. Em particular, blocos não verificam o número de argumentos.

```
# y não está definido, mas tudo bem
4.times {|x,y,z| puts "Oi, #{x} e #{y}"}
```

Um bloco passado como argumento de de uma função pode ser chamado por esta com o comando `yield`, eventualmente passando argumentos.

```
def ois
  yield("Fulano")
  yield("Ciclano")
  yield("Beltrano")
end
ois do
  |n| puts "Olá, #{n}"
end
```

A notação `&`*bloco* como argumento de uma função associa um bloco a uma variável, tornando-o um objeto completo. A classe deste objeto é *Proc* (ou procedimento).

```
def umm(&b)
  b.call
  yield
  puts b.class
end

umm {puts "ok"}
```

*Proc*s podem ser construídos com `Proc.new`.

Uma variante de Procs são *lambdas*. A diferença é que lambdas funcionam como funções, verificando os argumentos. Um `return` dentro de um lambda termina sua execução e retorna um valor, dentro de um Proc, o `return` termina a execução do contexto corrente. Para criar lambdas, usa-se `lambda` ou `->`.

```
triplo = ->(n){ 3*n }
triplo.call(2)
# ou
triplo.(2) # note o '.'
```

Para entender a diferença de comportamento de `return` nos dois casos, veja este exemplo.

```
def no_proc
  puts Proc.new { return "Saí do proc"}.call
  return "Resto do proc"
end

def no_lambda
  puts lambda { return "Saí do lambda" }.call
  return "Resto do lambda"
end

puts no_proc
puts no_lambda
```

Para passar um *lambda* como argumento de uma função que usa `yield` é preciso primeiro transformá-la em `Proc`, isto é feito com o operador `&`:

```
me = ->{puts "Mário!"}

def oi
  print "It's me, "
  yield
end

oi &me
```

Se não houver um `return` explícito, o último valor calculado é retornado.

4 Exemplos

Aqui estão alguns exemplos divertidos usando expressões regulares e elementos acima.

O operador `~` verifica se uma *string* bate com uma expressão regular. A negação é `!~`. Procure entender o restante.

Marcador de milhares

```
#!/usr/bin/env ruby

while x = gets
  break if x !~ /\d/
  'wtf' while x.sub!(/(.*\d)(\d\d\d)/,'\1.\2')
  puts ">> #{x}"
end
```

Histograma, com alguns truques:

```
#!/usr/bin/env ruby

val = Hash.new(0)                  #  valor default para novas entradas

while lin = gets
  lin.split.each { |p| val[p]+=1 }
end

val.sort_by {|k,v| -v}.each {|k,v| puts "%-20.20s : %4d\n"%[k,v]}
```

O que faz este código?

```
#!/usr/bin/env ruby
while l = gets.to_i
  break unless l > 1
  puts "Sim!" unless '1' * l =~ /^(11+)\1+$/;
end
```

5 Orientação a Objetos

A ideia básica é construir uma abstração, o centro da programação não é mais voltada para como definir a sequência de operações (programação imperativa), mas como modelar melhor as informações e dados tratados pelo programa. A partir de um bom modelo, a programação fica mais fácil.

Este tipo de abordagem traz outras vantagens, como reutilização de código, extensibilidade e independência de implementação.

O ponto de partida é o *objeto*, isto é, uma entidade representada no computador. As características básicas de um objeto são:

- Tudo é um objeto. Existem diversos *tipos*, ou *classes*, de objetos. Qualquer elemento de seu programa, incluindo o próprio, é representado como um objeto.
- Um programa é uma coleção de objetos interagindo entre si.
- Um objeto pode ser composto de outros objetos.
- Todo objeto tem um tipo. "Um objeto é uma instância de uma classe".
- Classes podem conter subclasses. (Um tipo é um conjunto, um subtipo é definido por restrições neste conjunto)
- Objetos da mesma classe reagem do mesmo modo.

Uma forma mais simples de descrever um objeto é indicar três propriedades:

- Estado. Descrito por um conjunto de variáveis.
- Comportamento. Descrito por funções, normalmente chamadas de métodos no linguajar de orientação a objetos.

- Identidade. Indicada por uma instância (em última análise, um endereço na memória).

Uma classe é representada no computador por uma *implementação*. Um conjunto de variáveis descreve o estado (como em uma *struct*) e um conjunto de funções descreve o comportamento.

Um paradigma destes implica em características muito especiais, tratadas nas seções seguintes.

5.1 Encapsulamento

Para que o objeto A atue sobre o objeto B, não é necessário que ele conheça a como B foi implementado. A única informação necessária é como chamar os métodos de B que fornecem as possíveis formas de interação. Na verdade, o implementador de A não deve fazer suposições sobre a implementação de B.

Toda classe pode esconder a implementação e apresentar apenas uma interface.

5.2 Herança

Uma classe pode ser apenas uma restrição sobre uma classe maior, um caso mais específico. Neste caso não há necessidade de se reconstruir toda a implementação: basta estender a implementação já existente da classe maior, restringindo o tipo.

Se B é uma classe derivada de A, B usa a mesma interface de A, talvez incluindo mais algumas coisas. O ponto importante é que em qualquer lugar do programa onde se usa um objeto do tipo A, pode-se também usar um objeto do tipo B sem a necessidade de reprogramação.

Vale a pena pensar um pouco mais sobre esta situação: B pode usar exatamente a mesma interface de A, mas mudando a implementação de algumas funções. Isto é chamado de sobrecarga (*override*).

Pode acontecer também de uma classe base apresentar uma função em sua interface, mas que não tem implementação alguma. É uma função virtual e que **deve** ser definida nas subclasses.

Existem duas formas de fazer herança: especializando (isto é, trocando métodos por outros mais específicos) ou estendendo (adicionando campos e métodos).

5.3 Polimorfismo

Funções que manipulam objetos de uma certa classe podem precisar atuar diferentemente se forem aplicadas a objetos de uma subclasse ou classes similares. Você, no programa, escreve a chamada do mesmo jeito para os dois casos, mas o código executado pode ser diferente.

Isto pode implicar em uma busca em tempo de execução (*late binding*), pois em algumas situações só é possível descobrir qual a função efetivamente chamada em tempo de execução.

Outra situação onde aparece o polimorfismo é em funções que recebem objetos como argumentos. Duas funções podem ter a mesma funcionalidade mas implementações diferentes, dependendo do tipo dos argumentos.

6 Implementando classes e objetos

Existem algumas diretrizes gerais para a implementação das classes. Estas diretrizes buscam manter o paradigma correto e garantir uma uniformidade na estrutura do código. Vamos discutir algumas delas, aproveitando para destacar alguns pontos importantes adicionais.

6.1 Agrupamento

Já vimos que classes podem ter uma relação hierárquica por meio de herança, mas esta não é a única possibilidade. Uma certa implementação pode precisar de classes auxiliares relacionadas e uma estrutura mais ampla do programa como um todo é interessante.

7 Classes

A definição de classes também segue uma sintaxe usual. Uma classe é composta de *atributos* e *métodos*. Atributos são campos e métodos são funções membro. Convenciona-se que todos os nomes de classes comecem com uma letra maiúscula.

```ruby
#!/usr/bin/env ruby
# coding: utf-8
class Carta
  def initialize(v,n)
    @naipe = n
    @valor = v
  end
  def to_s
    "%2s de %s"%[@valor,@naipe]
  end
end

class Baralho
  def initialize
    @cartas = []
    %w[♣ ♡ ♠ ♢].each do |n|
      (["A"] + Array(2..10) + ["J","Q", "K"]).each do |v|
        @cartas << Carta.new(v,n)
      end
    end
  end

  def print
    @cartas.each do |c| puts c ; end
  end

  def embaralha
    emb = []
    while @cartas.size > 0
      i = rand(0 ... @cartas.size)
      emb << @cartas.delete_at(i)
    end
    @cartas = emb
  end
```

```
  def distribui(nj, ncartas)
    return nil if nj * ncartas > @cartas.size

    mãos = Array.new(nj)
    mãos.map! {|p| p = Array.new}

    (1..ncartas).each do |i|
      (0...nj).each do |j|
        mãos[j] << @cartas.pop
      end
    end
    return mãos
  end
end

b = Baralho.new
b.embaralha
b.print
print("----" * 5)
puts
m = b.distribui(3,2)

(0...m.size).each do |i|
  puts "Cartas do jogador n. #{i+1}"
  m[i].each  do |c|
    puts c
  end
  puts '----'
end
```

No exemplo da classe `Carta`, não podemos mexer diretamente nos dois campos `naipe` e `valor` diretamente, como `c.valor=4` ou `puts c.naipe`. Para isso precisamos de métodos especiais para ler e/ou escrever nestas variáveis.

Estes métodos são chamados de *getter*s e *setter*s, respectivamente. Para fazer um *getter*, basta escrever um método com o nome do atributo, que retorna seu valor. O *setter* usa um método com o nome do atributo seguido de um `=`:

```ruby
class Carta
  def initialize(v,n)
    @naipe = n
    @valor = v
  end

  # getter
  def naipe
    @naipe
  end

  # setter
  def naipe=(n)
    @naipe = n
  end

  def to_s
    return "#{@valor} de #{@naipe}"
  end
end
```

Para gerar *getter*s e *setter*s automaticamente, basta colocar a expressão `attr_accessor` seguida com os nomes (símbolos) dos atributos.

```ruby
class Carta
  attr_accessor :naipe, :valor

  def initialize(v,n)
    @naipe = n
    @valor = v
  end

  def to_s
    return "#{@valor} de #{@naipe}"
  end
end
```

Para gerar apenas o *getter*, use `attr_reader`. Para *setter*, use `attr_writer`.

8 Herança

Veja este exemplo com herança simples. A classe `Progressão` define uma progressão genérica de números consecutivos. As outras classes implementam variações e aproveitam as propriedades de `Progressão` sem precisar redefini-las ou nem mesmo conhecer a implementação.

```
#!/usr/bin/env ruby
# coding: utf-8

class Progressão
  def initialize(ini=0)
    @prim = ini
    @atual = ini
  end

  def primeiro
    @atual = @prim
  end

  def prox
    @atual += 1
  end

  def imprime(n=1)
    print "#{@prim}"
    for i in 2..n
      print " #{prox}"
    end
  end

end
```

Neste exemplo, `require` faz com que o código de `Progressão.rb` seja lido e processado. Isto acontece uma única vez, mesmo que o comando aparecesse mais de uma vez no código. É equivalente ao `import` de Java e Python.

Progressão aritmética.

```ruby
#!/usr/bin/env ruby
# coding: utf-8

require './Progressão'

class Aritmética < Progressão
  def initialize(inc = 1, ini)
    @inc = inc
    super(ini)
  end

  def prox
    @atual += @inc
  end
end
```

Progressão geométrica.

```ruby
#!/usr/bin/env ruby
# coding: utf-8

require './Progressão'

class Geométrica < Progressão
  def initialize(inc = 1, ini)
    @inc = inc
    super(ini)
  end

  def prox
    @atual *= @inc
  end
end
```

Fibonacci.

```
#!/usr/bin/env ruby
# coding: utf-8

require './Progressão'

class Fibonacci < Progressão
  def initialize(v1 = 0, v2 =1)
    @ant = v1
    @atu = v2
    super(v2)
  end

  def prox
    @ant, @atu = @atu, @atu + @ant
    return @atu
  end
end
```

Uma vez definidas as classes, elas podem ser usadas normalmente. Veja neste exemplo que o vetor`vários` usa as sequências uniformemente, afinal todas são `Progressão`.

```
#!/usr/bin/env ruby
# coding: utf-8

require './Arit'
require './Geom'
require './Fibo'

a = Aritmética.new(2,7)
g = Geométrica.new(4,5)
f = Fibonacci.new(1,1)

def linha()  puts "\n" + '-'*20 ; end

a.imprime(20)
linha
g.imprime(4)
linha
f.imprime(5)
linha
```

```
vários = [
  Aritmética.new(1, 6),
  Progressão.new(2),
  Fibonacci.new(1, 1),
  Geométrica.new(1, 2),
  Fibonacci.new(7, 9),
  Geométrica.new(3, 3),
  Progressão.new(4),
  Geométrica.new(5, 1),
  Geométrica.new(2, 2.3),
]


def moldura
  print "["
  yield
  puts "]"
end

vários.each {|p| moldura {p.imprime(5)}}
```

9 Herança Múltipla (ou não)

Para criar um objeto com características de dois tipos diferentes precisaríamos ter herança múltipla (a classe extenderia mais de uma classe base).

No entanto herança múltipla tem seus inconvenientes. Uma classe pode herdar implementações diferentes dos mesmos métodos ou ainda ser derivada mais de uma vez da mesma base. Existe ainda o problema de um campo ou método de um ancestral comum ser modificado indiretamente por métodos diferentes.

Vejamos um exemplo onde aparece a necessidade de herança múltipla: considere a hierarquia de alimentos da figura 1.

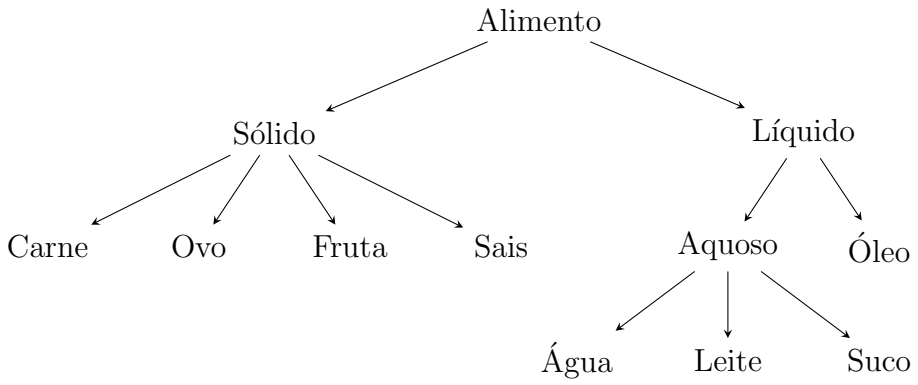

Figura 1: Hierarquia de Alimentos

Esta hierarquia pode ser facilmente estendida para vários casos. No entanto, temos um problema se precisarmos escolher alimentos de origem animal, mineral ou vegetal. Existem exemplos em diversos ramos desta árvore.

Uma solução, sem o fazer uso de herança múltipla, são *mixins*. *Mixins* declaram métodos que podem ser usados em outras classes. Em Ruby, os *mixins* são definidos por `modules`. Instâncias de uma classe `C` que incluem um `module M` recebe os métodos (portanto o comportamento) definidos em `M`.

Este arranjo simples evita os problemas de herança múltipla mas fornece um mecanismo para que objetos compartilhem propriedades de outro tipo, além do esquema de derivação usual.

Podemos colocar métodos adicionais nos objetos, fazendo com que eles se comportem como elementos de um tipo adicional. Isto é chamado de *duck typing* (se anda como um pato e grasna como um pato, deve ser um pato). Com módulos, isso fica mais fácil de fazer.

Os módulos também podem ser usados para simplesmente agrupar métodos e variáveis dentro de um grupo separado, com um espaço de nomes próprio.

```
#!/usr/bin/env ruby
# coding: utf-8

module Fritável
  @frito = false
  def fritar
    print self.class.name
    if @frito
      puts " >>> Torrou!!"
    else
      puts " > shhhhhhhhzz!"
      @frito = true
    end
  end
end

class Alimento
end

class Sólido < Alimento
end

class Carne < Sólido
  include Fritável
end

class Ovo < Sólido
  include Fritável
end

class Fruta < Sólido
end
```

```
class Sais < Sólido
end

class Líquido
end

class Aquoso < Líquido
end

class Agua < Aquoso
end

class Leite < Aquoso
end

class Suco < Aquoso
end

class Oleo < Líquido
  include Fritável
end

perrier = Agua.new

bife = Carne.new
bife.fritar
codorna = Ovo.new
codorna.fritar

bife.fritar
```

Um modulo pré-definido muito interessante é o `Comparable`. Um dos operadores de Ruby é o comparador total[2]: `<=>`, que retorna -1,0 ou 1 se os valor da esquerda é menor, igual ou maior do que o valor da direita. Implementando este operador na sua classe e incluindo o módulo `Comparable`, todas os outros operadores de comparação ficam automaticamente definidos. O exemplo a seguir foi retirado de `ruby-doc.org`.

[2]Em Perl esse operador é conhecido como "disco voador"

```
class SizeMatters
  include Comparable

  def <=>(other)
    str.size <=> other.str.size
  end
  def initialize(str)
    @str = str
  end
  def inspect
    @str
  end
end

s1 = SizeMatters.new("Z")
s2 = SizeMatters.new("YY")
s3 = SizeMatters.new("XXX")
s4 = SizeMatters.new("WWWW")
s5 = SizeMatters.new("VVVVV")

s1 < s2                         #=> true
s4.between?(s1, s3)             #=> false
s4.between?(s3, s5)             #=> true
[ s3, s2, s5, s4, s1 ].sort     #=> [Z, YY, XXX, WWWW, VVVVV]
```

10 Exceções e Tratamento de Erros

Um acontecimento inesperado é chamado de *exceção* no jargão de orientação a objetos. Normalmente uma exceção é associada a algum erro, mas pode significar qualquer evento fora do normal, como uma entrada inválida.

Como em geral existem várias chamadas aninhadas de métodos pertencentes a objetos diferentes, o tratamento direto de um erro fica dificultado. Por exemplo, ao se tentar abrir um arquivo, uma exceção pode acontecer em qualquer nível: o arquivo pode não existir, pode existir mas não se tem permissão para abri-lo, pode estar em uma unidade remota e a rede cair, pode não haver memória para tratar o arquivo, etc. Cada um destes erros é identificado em uma classe diferente, mas o tratamento deve ser feito pelo programa do usuário, não necessariamente pelas bibliotecas que interagem com o sistema operacional.

Uma exceção é uma classe que representa o erro ou evento anormal. Pode-se criar novas exceções se necessário. As principais exceções padrão do Ruby são:

- NoMemoryError
- ScriptError
  - LoadError
  - NotImplementedError
  - SyntaxError
- SignalException
  - Interrupt
- StandardError
  - ArgumentError
  - IOError
    - EOFError
  - IndexError
  - LocalJumpError
  - NameError
    - NoMethodError
  - RangeError
    - FloatDomainError
  - RegexpError
  - RuntimeError
  - SecurityError
  - SystemCallError
  - SystemStackError
  - ThreadError
  - TypeError
  - ZeroDivisionError

Quando um método encontra uma anormalidade, ele pode "levantar" uma exceção com o comando `raise`. A exceção pode ser capturada em algum lugar na pilha de chamadas e ser tratada de acordo. Se a exceção não for capturada em momento algum, ela é apresentada como uma mensagem de erro na saída.

```
puts 3/0
ZeroDivisionError:  divided by 0
        from (irb):3:in '/'
        from (irb):3
        from /usr/bin/irb:11:in '<main>'
```

Para capturar uma exceção no código, é preciso identificar o bloco que se quer proteger. É o equivalente aos blocos `try/catch` de outras linguagens. Em Ruby, marca-se o início do bloco com `begin` e a região do tratamento com `rescue`. A estrutura geral é esta:

- `begin` seguido pelo código "protegido", onde as exceções podem ser lançadas.
- `rescue` ou `rescue => e` ou `rescue` *exceção* `=> e` seguido pelo tratamento da exceção — pode aparecer várias vezes, para especificar tratamentos de exceções diferentes
- `else` seguido tratamento para o caso de exceções não especificadas. Este bloco é opcional.
- `ensure` seguido por código que deve ser executado sempre, independente do acontecimento de uma exceção. Também é opcional.
- `end`

Dentro de um bloco `rescue` ou `else`, o comando `retry` faz com que o trecho todo seja reexecutado.

```
begin
  puts "Entre com um número negativo"
  # chomp elimina o \n no final
  x = gets.chomp.to_i
  raise "Eu disse negativo" if x >= 0
  puts "Muito bem!!!"
rescue =>e
  # e.message também é colocada em $!
  puts e.message
  retry
end
```

O comando `raise` pode ser escrito de maneiras diferentes:

```
# usando RunTimeError como exemplo
raise RunTimeError.new("Mensagem de erro")

# ou
raise RunTimeError,  "Mensagem de erro"

# no caso específico de RunTimeError, também
# podemos escrever simplesmente
raise "Mensagem de erro"
```

De modo similar, `rescue` também possui várias formas:

```
# Para pegar qualquer exceção (não aconselhável)
rescue

# Para capturar a exceção na variável v
# neste caso, a exceção é do tipo StandardError
rescue =>v

# Para capturar uma exceção específica
rescue ArgumentError

# ou ainda
rescue ArgumentError => v
```

10.1 *Throw* e *Catch*

O comando `catch` serve para anotar um bloco com um nome (*label*, na verdade um símbolo), que será terminado se um `throw` para aquele *label* for executado. É um pulo direto para o final de um bloco acima na pilha de execução.

Serve para interromper o processamento dentro de um aninhamento grande de chamadas e cair fora rapidamente. O processamento de `throw` e `catch` é muito mais rápido do que exceções.

Este é um exemplo simples que achei na *web*

```
def Entrada(prompt)
  print prompt
  l = gets.chomp
  throw :refaz if l == '!'
  return l
end

catch :refaz do
  nome  = Entrada 'Nome: '
  idade = Entrada 'Idade: '
  sexo  = Entrada 'Sexo: '
  puts "Registro completo"
end
```

Usar '!' só funciona uma vez, pois depende da pilha.

`throw` também pode especificar o valor de retorno do bloco `catch` correspondente, se colocarmos um segundo parâmetro. Veja este outro exemplo, adaptado de
`https://jacopretorius.net/2012/01/catch-and-throw-in-ruby.html`

```
random_numbers = catch (:random_numbers) do
  result = []
  10.times do
    num = rand(100)
    throw(:random_numbers, []) if num % 7 == 0
    result << num
  end
  result
end

puts random_numbers
```

Nele, garantimos que o vetor não tem múltiplo de 7. Se aparecer algum, o vetor será vazio.

Pense em resolver as 8 rainhas em Ruby.

11  Metaprogramação

Podemos entender Metaprogramação como programação do programa, ou seja, é possível estender o código em tempo de execução.

Por ser uma linguagem de *script* interpretada, Ruby é capaz de introspecção,, ou seja, o programa tem acesso à sua própria estrutura de dados, como nomes de variáveis, espaços de nomes, etc. Com o auxílio de alguns métodos especiais e recursos sintáticos, é possível criar código dinamicamente.

Blocos, por exemplo, por serem trechos de código, podem ser "injetados" em outras funções. Além disso, blocos, *procs* e *lambdas* são *closures* — capturam o escopo do momento em que foram criados. Isso permite criar funções que geram funções.

```
def mold(base, tam)
  lin = base*tam
  res = lambda {|x|
    puts lin
    yield x
    puts lin
  }
end

f = mold('-',12) {p "oi"}
f.call(90)
g = mold('=',45) {|x| p x*50}
g.call(90)
```

Isso ainda não é metaprogramação, mas é um recurso que ajuda muito a fazer metaprogramação.

11.1 Introspecção

O cerne da metaprogramação é a introspecção, a capacidade de observar a própria estrutura.

Dado um objeto, podemos perguntar a ele quais são os métodos que ele "atende", qual sua classe e superclasses, os nomes dos atributos (variáveis de instância) de uma classe, etc. Aliás, uma classe também é um objeto!

Existem diversos métodos especiais para consultar objetos e classes. Estes são alguns exemplos.

```
42.respond_to? :times
42.respond_to? :odd?
42.respond_to? :nome_inventado
```

Podemos perguntar a hierarquia de classes e módulos:

```
String.ancestors
42.class
Integer.superclass
```

Ou verificar quais os métodos e atributos:

```
String.methods
42.methods
Integer.superclass
class Algo
  def initialize(n)
      @v = n
      @outra = 0
  end
end
x = Algo.new(2)
x.instance_variables
```

A rigor, toda chamada de método é uma mensagem ao objeto dizendo a ele o que deve ser feito. Isso aparece de forma bem explícita em Ruby:

```
class C
  def grite
     puts 'AAAAARRRGHYOOOWWWLLL'
  end
end

c = C.new
c.grite
c.send 'grite'
c.send :grite
g = "grite"
c.send(g)       # ou c.send g
```

11.2 Classes abertas

Em Ruby, as classes são abertas. Podemos mexer na implementação a qualquer momento:

```
class Array # classe padrão
  def não
      puts "no es possible!"
  end
end

[].não
```

Veja que isso não é herança, mas alteração da própria classe.

11.2.1 Objetos abertos?

Quando uma instância é criada, ela ganha uma classe adicional só para ela. Essa nova classe é inserida na hierarquia entre a instância e a classe mãe e é chamada de *singleton*.

Podemos alterar essa classe, fazendo com que uma *instância* ganhe métodos novos.

```
class A
end

a = A.new
b = A.new

def b.oi
   puts "Hey"
end

a.oi
b.oi
```

Uma outra característica é permitir que um bloco seja executado dentro do escopo de uma instância ou de uma classe. Dessa forma, o bloco tem acesso ao estado interno e pode modificar a classe ou instância ao ser executado. Ao invés de um bloco, pode ser usada uma *string* que será interpretada.

```
class A
  def initialize
     @um = 42
  end
  def um
     @um
  end
end

a = A.new
a.instance_eval { puts 2*num }
a.instance_eval { puts 2*@num }
a.instance_eval "puts 2*num"
a.instance_eval "def uia ; puts 'caramba' ; end"
a.uia
```

De forma similar `class_eval` executa o bloco no escopo da classe. Uma definição dentro do bloco será visível por todas as instâncias.

Naturalmente, usar qualquer forma de `eval` com *strings* pode ser muito perigoso.

11.3 Criação dinâmica de métodos

Com o que foi apresentado nas seções anteriores podemos criar métodos para instâncias e classes, mas existe uma forma mais compacta.

Para criar um método para instâncias dinamicamente, pode-se usar `define_method`, que é um método *privado*.

```
class A
  # só uma forma diferente de definir um método usual
  define_method(:ok) {puts "normal"}

  # criação dinâmica
  def cria(s)
    A.define_method(s) {puts "eu sou #{s}"}
  end
end

a = A.new
a.ok
a.cria("oi")
a.oi
```

Da forma em que está escrita, a função `cria` está diretamente amarrada à classe `A`. Isso é muito ruim se usarmos herança. É melhor fazer desta forma:

```
class A
  # só uma forma diferente de definir um método usual
  define_method(:ok) {puts "normal"}

  # criação dinâmica
  def cria(s)
    # pegamos a classe o objeto atual
    self.class.define_method(s) {puts "eu sou #{s}"}
  end
end
```

Vejamos uma variação mais interessante:

```
class Shell
  def cria(s)
    self.class.define_method(s) do
      |*x| # array com todos os argumentos
      x = [] unless x
      parms = x.join(" ")
      `#{s} #{parms}`
    end
  end
end

a = Shell.new
a.cria('ls')
a.ls
a.ls('*.rb')
```

Uma aplicação real, que já vimos, é `attr_accessor` e suas variações.

De forma similar, existe o `define_singleton_method`, cujo comportamento é similar.

Finalmente, há o `method_missing`, que é chamado toda vez que um método não é encontrado. `method_missing` recebe três parâmetros: o nome do método, os argumentos passados (se houver) e o bloco (se houver).

```
class Ranzinza
  def method_missing(met,*parms, &bloco)
      puts "Não tenho ideia do que você quer dizer com #{met}"
  end
end

r = Ranzinza.new
r.cabum
```

Neste exemplo, não parece ser muito útil, mas podemos criar o método faltante ou delegar para outra classe ou objeto que o implemente, usando `send`.

11.4 Cuidados

O uso de metaprogramação implica em avaliação e busca de métodos em tempo de execução, isso tem um custo adicional significativo. O abuso da técnica tornará o código bem mais lento. No entanto, há um ganho significativo da legibilidade se usada corretamente.

Por outro lado, a legibilidade pode ficar bastante complicada se o uso da metaprogramação não for feita corretamente. O principal objetivo é evitar repetições e ser usada para deixar o programa mais claro. A responsabilidade é de quem programa.

O uso inadequado também pode introduzir problemas de segurança, com tantos `evals`.

A regra geral é "use com moderação e inteligência". Não abuse.

Índice Remissivo