

Brincando com Ruby

Vítor Baptista
vitor@vitorbaptista.com



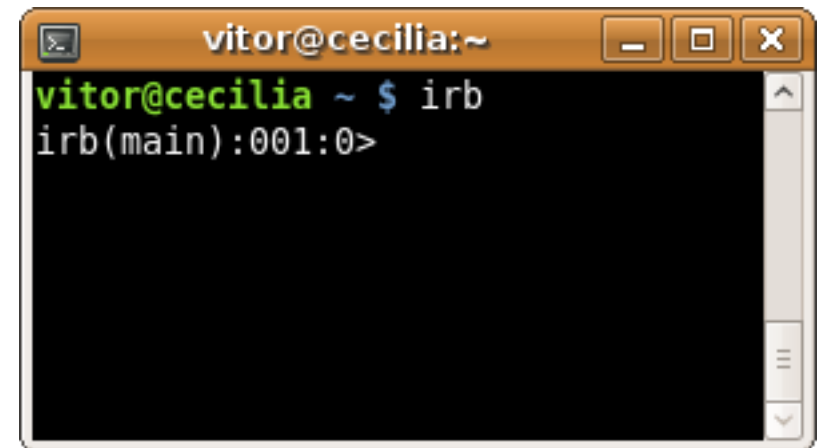
Ruby?

- Criada em 1995;
- Yukihiro “matz” Matsumoto;
- Totalmente orientada a objetos (não existem tipos primitivos);
- Interpretada;
- Muito influenciada por Smalltalk e Perl;
- Dinâmica (tudo pode mudar em tempo de execução).



IRB

- Interactive Ruby;
- É um shell que executa código Ruby;
- Pode-se definir classes, métodos, funções e todas declarações Ruby.

A screenshot of a terminal window titled "vitor@cecilia:~". The window shows the command "irb" being executed, resulting in the prompt "irb(main):001:0>". The terminal has a black background and white text. The window title bar is orange and contains the text "vitor@cecilia:~" and standard window control buttons (minimize, maximize, close).

```
vitor@cecilia ~ $ irb
irb(main):001:0>
```

Começando a brincar

numero = 30

=> 30

outro_numero = -20

=> -20

numero + outro_numero

=> 10

numero + outro_numero.abs

=> 50

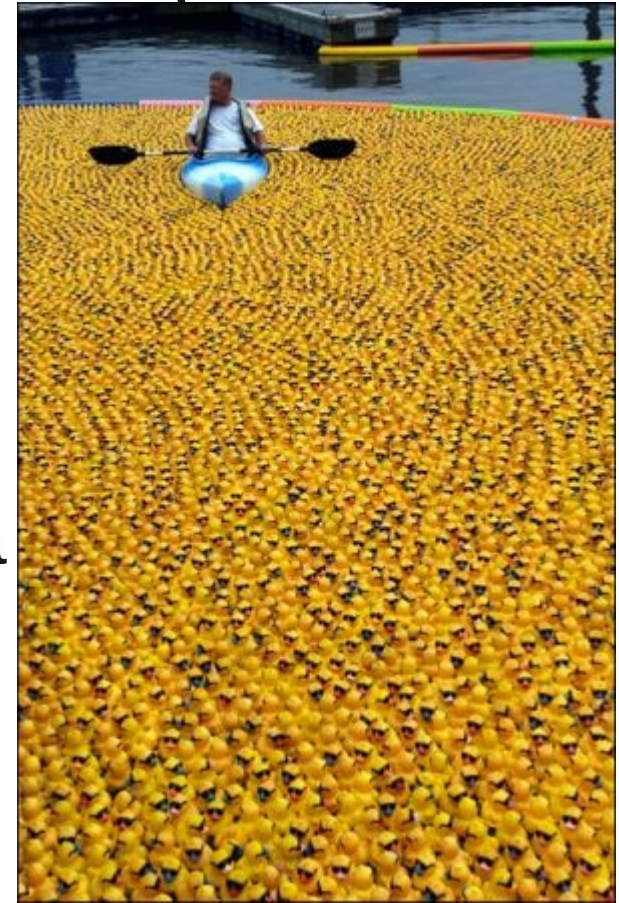
Primeiras três regras de Ruby

- Não se usa “;” para indicar o fim da linha;
- Variáveis não têm tipo;
- Quando damos nomes a variáveis, as boas maneiras regem que separemos nomes com “_”.

Mas não é ruim não ter tipos?

- Não importa o tipo do objeto. O que importa são as mensagens (métodos?) que ele é capaz de responder;

- Duck Typing - se ele grasna, nada e voa feito um pato, então para mim é um pato.



Definindo um método

```
def soma(primeiro, segundo)  
  primeiro + segundo  
end
```

```
soma(10, 50)  
soma 50, 90
```

```
# => 60  
# => 140
```

Mais quatro regrinhas

- Um método é definido com **def**, seu nome e parâmetros entre parênteses;
- Blocos de código (métodos, classes, “ifs”) terminam com **end**. Nada de “{}”;
- Usar **return** é opcional (normalmente não é usado). A última expressão avaliada é retornada;
- Os parênteses na chamada dos métodos não são obrigatórios, mas cuidado com a legibilidade.

Definindo uma classe

```
class Musica
  def initialize(nome, artista, duracao)
    @nome = nome
    @artista = artista
    @duracao = duracao
  end
end
```

```
musica = Musica.new "Abaladinha", "Molestrike", 4
# => #<Musica:0xb7cfaeb0 @artista="Molestrike",
@nome="Abaladinha", @duracao=4>
```

Mais regras

- A definição de uma classe começa com **class**;
- Classes são sempre abertas. Você pode redefinir os métodos em qualquer lugar, é só declará-la novamente;
- Nomes de classe normalmente são definidos usando **CamelCase** (ex.: **NomeDeClasse**).

Atributos

```
class Musica
```

```
  attr_writer :duracao
```

```
  attr_reader :duracao
```

```
  attr_accessor :nome
```

```
end
```

```
musica = Musica.new("Acontece", "Cartola", 110)
```

```
musica.duracao = 118
```

```
musica.nome = "Ensaboa Mulata"
```

Regras de novo

- **attr_reader** define métodos de acesso a um atributo de instância de um objeto (um getter);
- **attr_writer** define métodos de alteração (um setter);
- **attr_accessor** faz o mesmo que os dois juntos.

Atributos virtuais

```
class Musica
  def duracao_em_minutos
    @duracao/60.0
  end

  def duracao_em_minutos=(nova_duracao)
    @duracao = (nova_duracao * 60).to_i
  end
end
```

O que foi isso?

- Criamos métodos de acesso a um atributo, mas o cliente não sabe se são métodos ou se ele está acessando os atributos diretamente;
- Sobrescrevemos o operador “=” (atribuição) para a nossa propriedade. Ruby tem sobrecarga de operadores (só alguns)!

Constantes

```
class ListaDeMusicas
  DURACAO_MAXIMA = 5*60 # 5 minutos
  DURACAO_MINIMA = 1*60 # 1 minuto
end
```

Variáveis de classe

```
class ListaDeMusicas  
  @@total_de_chamadas  
end
```

Métodos de classe

```
class ListaDeMusicas
  def ListaDeMusicas.e_muito_grande?(musica)
    musica.duracao > DURACAO_MAXIMA
  end

  def self.e_muito_pequena?(musica)
    musica.duracao < DURACAO_MINIMA
  end
end
```

E aí?

- Variáveis de classe são definidas com dois arrobas (@@);
- Constantes são definidas com seu nome em **CAIXA_ALTA**;
- Métodos de classe são definidos colocando o nome da classe antes do nome do método (ou usando **self** (é o **this** de Ruby))

Controle de acesso

```
class MinhaClasse
  def metodo1
  end
  protected
  def metodo2
  end
  private
  def metodo3
  end
  public
  def metodo4
  end
end
```

O padrão é public

Como acessar?

- Os níveis de acesso são definidos em blocos.
Por padrão é **public**;
- Um bloco termina onde outro começa.

Arrays

```
a = [1.618, "Sam", 42] # => [1.618, "Sam", 42]
a[0] # => 1.618
a[3] # => nil
a << 20_04_89 # Adiciona ao fim do array
a[4] = 115 # Adiciona outro elemento
a[-1] # => 115
a[2..4] # => [42, 200489, 115]
```

Explicando arrays

- Arrays são definidos com “a = []” ou “a = `Array.new`”;
- São acessados pelo índice, podendo-se usar números negativos para percorrê-lo de trás para frente;
- Quando se acessa um índice que não existe, retorna `nil`.

Explicando arrays

- Novos itens podem ser adicionados usando o próprio operador [], como em “a[4] = 121”;
- Para adicionar um item ao fim do array, usa-se o operador <<, como em “a << 51”;
- Arrays podem ser fatiados como em “a[1..3]”, que retorna um novo array com os valores do índice 1 ao 3 de a, ou “a[1...3]”, que retorna de 1 ao 2 (perceba os três pontos);
- Arrays podem ser tratados como listas.

Hashes

```
h = { 'potoff' => 'vodka', 'jureminha' => 'cana',  
      'carreteiro' => 'vinho' }
```

h.length	# => 3
h['potoff']	# => 'vodka'
h['jureminha']	# => 'cana'
h['dreher'] = 'conhaque'	# => 'conhaque'
h.length	# => 4
h[51] = 'cana'	# => 'cana'
h.length	# => 5
h['potoff'] = 13	# => 13

Entendendo hashes

- São conjuntos chave-valor (lembra do Map em Java?);
- Declarados com “h = {}” ou “h = Hash.new”;
- Para adicionar itens, basta usar o operador [], como em “h['chave'] = 'valor'”;

Você consegue explicar isto?

```
class ListaDeMusicas
  def com_titulo(titulo)
    for i in 0...@musicas.length
      return @musicas[i] if titulo == @musicas[i].nome
    end

    return nil
  end
end
```

E isto?

```
class ListaDeMusicas
  def com_titulo(titulo)
    @musicas.find { |musica| titulo == musica.nome }
  end
end
```

Blocos!

- São pedaços de código que podem ser passados como parâmetros para funções, normalmente para fazer filtragem, ordenação, etc.

- Você pode definir seus próprios métodos que aceitam blocos.



Como?

```
def fib_up_to(max)
  i1, i2 = 1, 1          # atribuição paralela
  while i1 <= max
    yield i1
    i1, i2 = i2, i1 + i2
  end
end

fib_up_to(1000) { |f| print f, ' ' }
```

Yield

n **1** rendimento, lucro, produto. **2** produção. • *vt+vi*
1 dar, conceder, consentir, permitir, aquiescer, autorizar. **2** entregar(-se), capitular, render-se, deixar (para o inimigo). **3** ceder (pressão, peso). **4** render, produzir.

Strings

'escape usando “\”’ # => escape usando “\”

'That's right!' # => That's right

"Segs/dia: #{24*60*60}" # => Segs/dia: 86400

'a' << 'b' << 'c' # => abc

```
def to_s
  "Música: #{@musica} Artista: #{@artista}"
end
```

O que são strings?

- Sequências de caracteres de 8 bits;
- Definidas usando " (aspas simples) ou "" (aspas duplas);
- Usando aspas duplas, pode colocar expressões ou variáveis, como em “#{6 * 9}” ou “#{@musica}”.

Ranges

`1..10` # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

`1...10` # [1, 2, 3, 4, 5, 6, 7, 8, 9]

`'a'..'z'` # alfabeto

`array = [1, 2, 3]`
`0..array.length` # [0, 1, 2, 3]

`(1..10).to_a` # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

O que são ranges?

- Conjunto de objetos em sequência, normalmente caracteres ou números;
- Definidos usando, por exemplo **1..10** (inclusive) ou **1...10** (exclusive);
- Você pode definir seus próprios ranges com o operador “<=>” (disco voador?).

Criando seu próprio range

```
class Volume
  include Comparable

  attr :volume
  def initialize(volume)
    @volume = volume
  end
  def to_s
    '#' * @volume
  end
  def <=>(other)
    self.volume <=> other.volume
  end
  def succ
    Volume.new(@volume.succ)
  end
end
```

Números

```
num = 42
6.times do
  puts "#{num.class}: #{num}"
  num *= num
end
```

Um pouco mais sobre métodos

```
def cara_legal(arg1 = "Russo", arg2 = "Guilmour", arg3 = "Gil")  
  "#{arg1}, #{arg2}, #{arg3}"  
end
```

```
cara_legal
```

```
# => "Russo, Guilmour, Gil"
```

```
cara_legal("Maddog")
```

```
# => "Maddog, Guilmour, Gil"
```

```
cara_legal("Maddog", "Stallman")
```

```
# => "Maddog, Stallman, Gil"
```

```
cara_legal("Maddog", "Stallman", "Torvalds")
```

```
# => "Maddog, Stallman, Torvalds"
```

Um pouco mais sobre métodos

```
def apocalipse
  ano_atual = 2009
  while (true)
    return "Apocalipse", ano_atual if (ano_atual % 6*6*6) == 0
    ano_atual += 1
  end
end
```

```
apocalipse # => ["Apocalipse", 2010]
```

Um pouco mais sobre métodos

```
def tres(a, b, c)  
  "Recebi #{a} #{b} #{c}"  
end
```

```
tres(1, 2, 3)
```

```
# => "Recebi 1 2 3"
```

```
tres(1, *['a', 'b'])
```

```
# => "Recebi 1 a b"
```

```
tres(*(19..21).to_a)
```

```
# => "Recebi 19 20 21"
```

Ainda em métodos

```
def buscar(nome, params)
  # Os parâmetros são acessados como em
  # params[:titulo], params[:autor], etc.
  # i.e.: params é um hash
end

buscar('musicas', :titulo => 'Trenzinho Caipira',
      :autor => 'Villa-Lobos')
```

if e unless

```
puts 'seria' if sesse
```

```
print total unless total.zero?
```

```
if x > 0
```

```
  puts x
```

```
elsif x < 0
```

```
  puts "#{x} é menor que zero"
```

```
else
```

```
  puts "x é igual a zero"
```

```
end
```

Case/when (vulgo: switch)

```
bissexto = case
  when ano % 400 == 0: true
  when ano % 100 == 0: false
  else ano % 4 == 0
end
```

while

```
puts "Olá!" while false
```

```
begin
```

```
  puts "Tchau!"
```

```
end while false
```

```
while x < 10
```

```
  puts "x é #{x}"
```

```
  x = x + 1
```

```
end
```

Quem precisa de um for?

```
3.times do  
  print "Ho! "  
end
```

```
0.upto(9) do |x|  
  print x, " "  
end
```

```
(1..5).to_a.each { |n| puts "#{n} " }
```

Mas você ainda quer um for?

```
for i in ['fee', 'fi', 'fo', 'fum']  
    print i, " "  
end
```

```
for i in 1..3  
    print i, " "  
end
```

Break, redo e next

```
while linha = gets
  next if linha =~ /^s*#/
  break if linha =~ /^END/
  redo if linha.gsub!(/`(.*)`/) { eval($1) }
end
```

Tente outra vez

```
for i in 1..100
  print "Agora em #{i}. Recomeçar? (s/n) "
  retry if gets =~ /^s/i
end
```

Acabou!

(A primeira parte)

Dúvidas?

Referências

Até aqui, estes slides foram baseados (quase copiados) da apresentação Aprendendo Ruby e Rails do Maurício Linhares (Valeu, Maurício!)

Que, por sua vez, foi completamente, absurdamente, absolutamente, plagiadamente baseado em **Programming Ruby – The Pragmatic Programmer's Guide 2nd Ed.**

Brincando com Ruby!



Gosu



Biblioteca criada em 2001 para desenvolvimento de jogos 2D (3D também) multiplataforma.

Feita em C++. Pode usá-la programando em C++ ou Ruby.

O que vamos fazer?



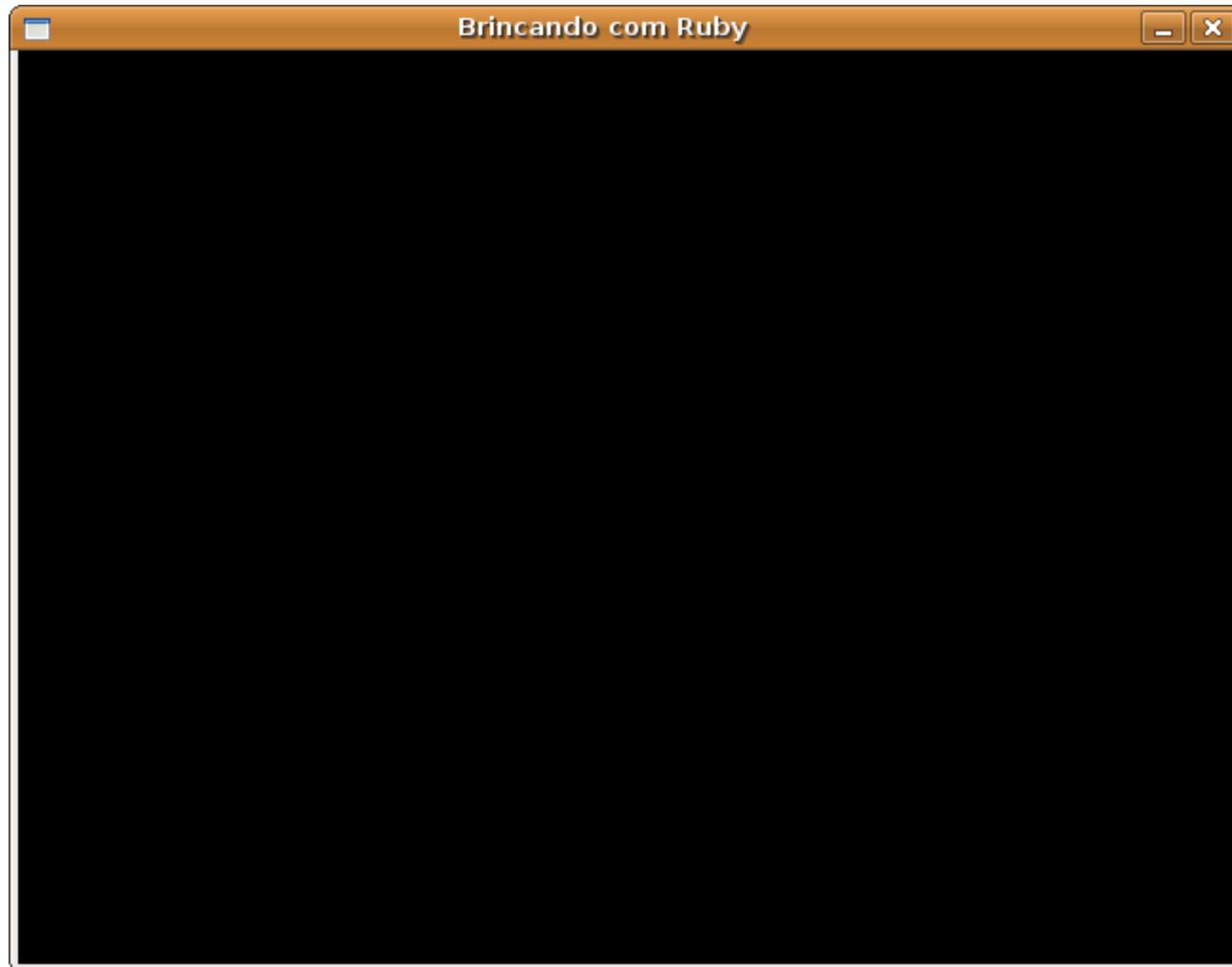
Primeiro passo

```
mkdir brincando  
cd brincando
```

```
cp -r /var/lib/gems/1.8/gems/gosu-0.7.11/examples/* .
```

```
ruby Tutorial.rb
```

Criando uma janela



Modificando o background



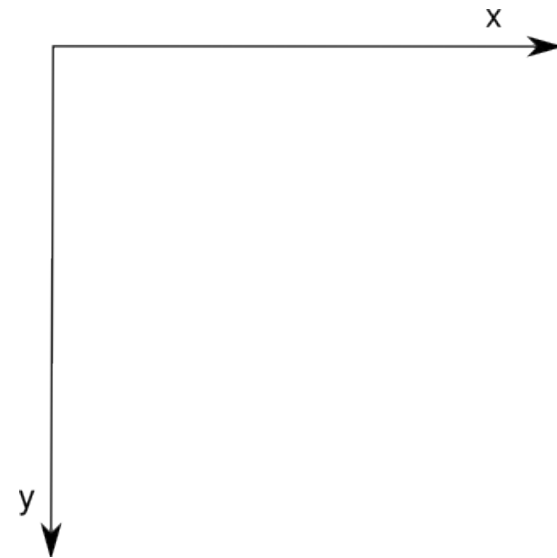
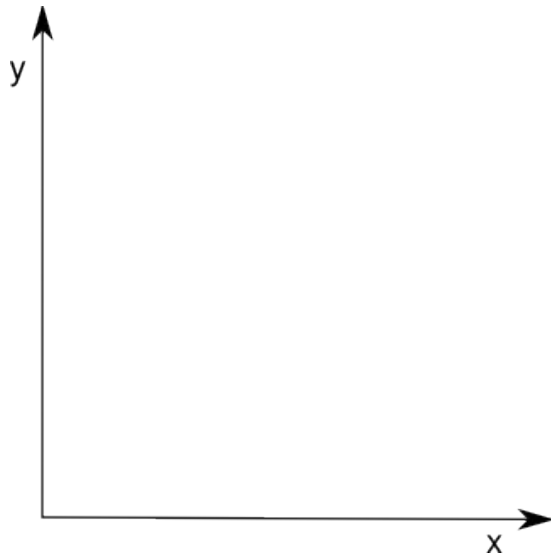
Adicionando um jogador



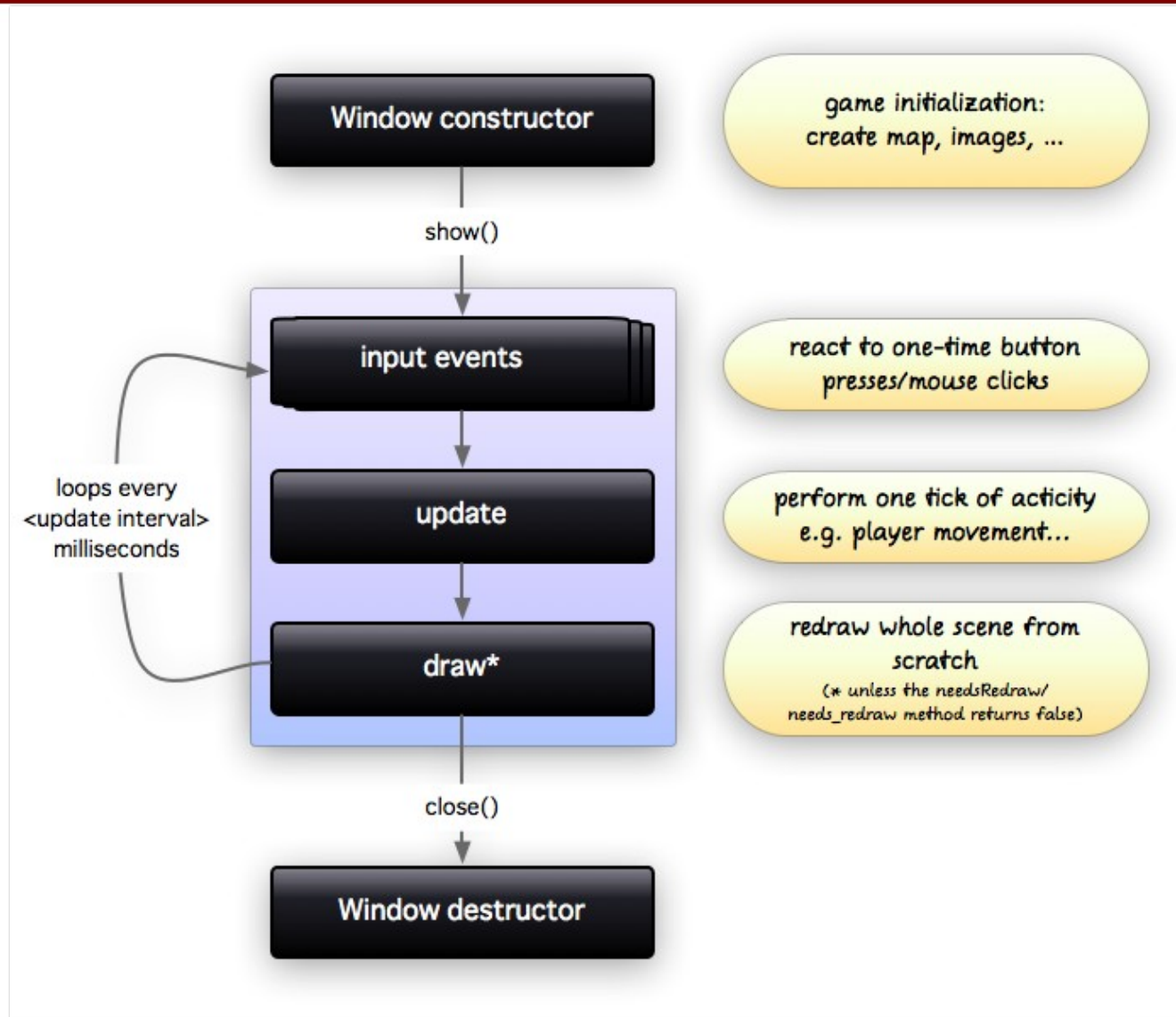
Warp



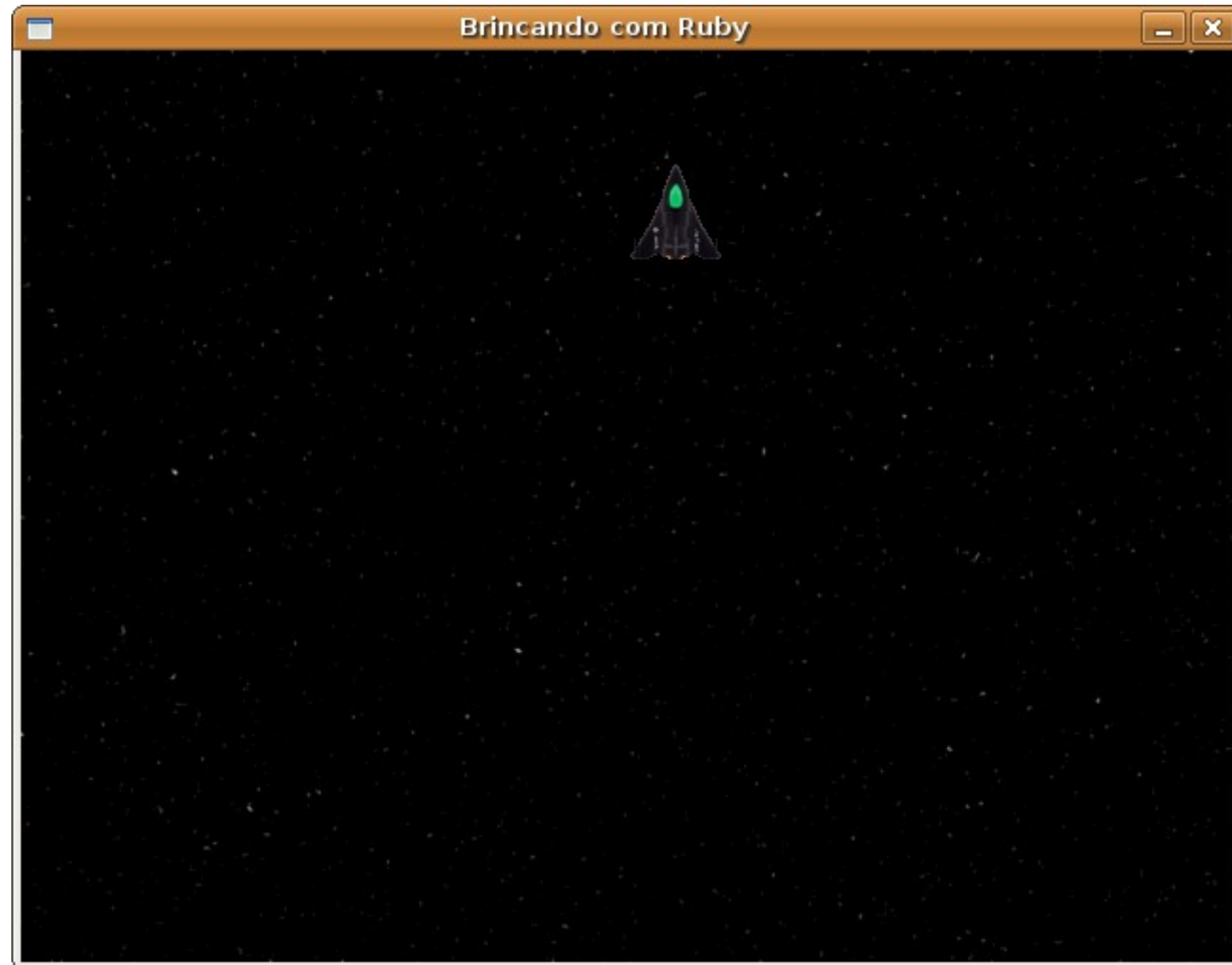
E essas coordenadas?



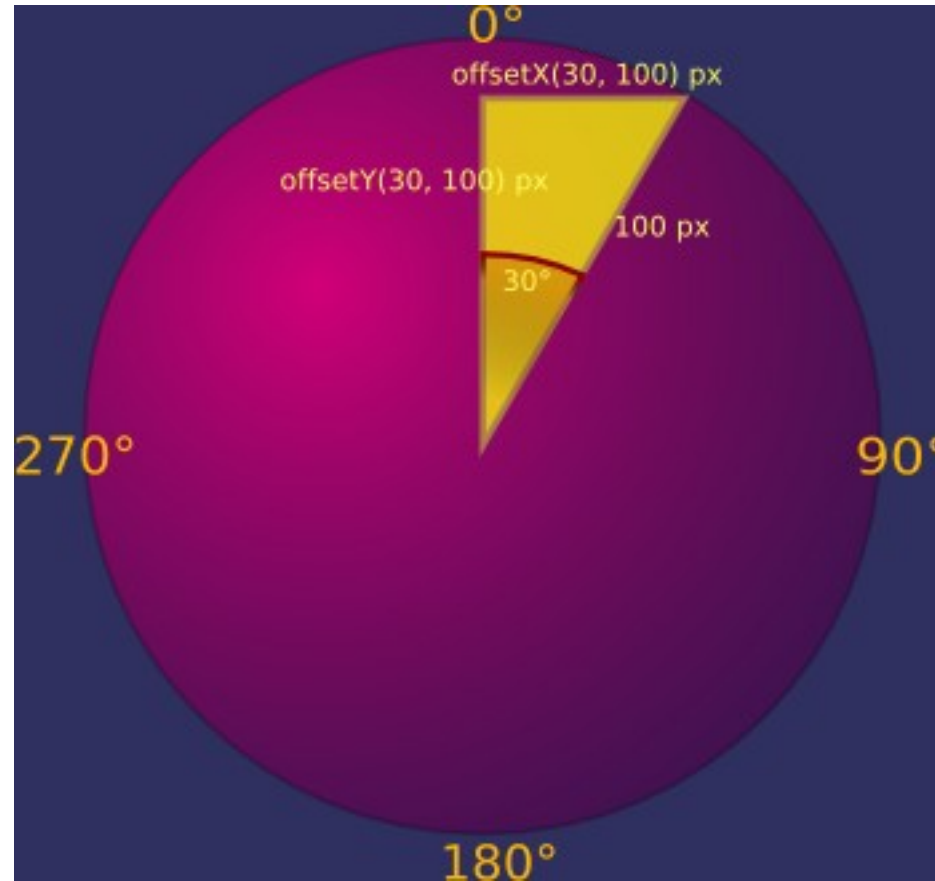
Como funciona o Gosu?



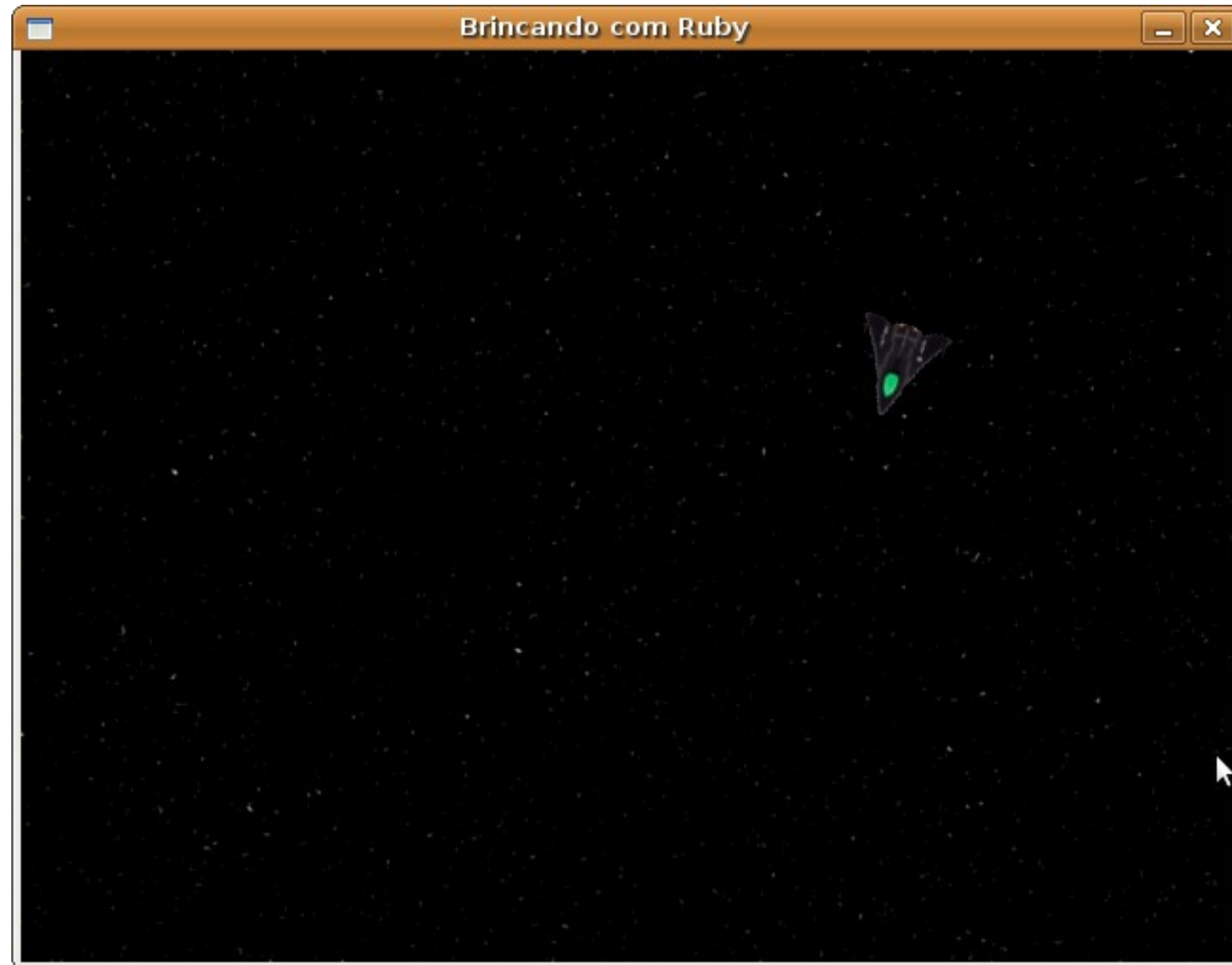
Movimentação básica



Um pouco de matemática



Movimentando direito



Lucy in the sky with diamonds



Lucy in the sky with diamonds



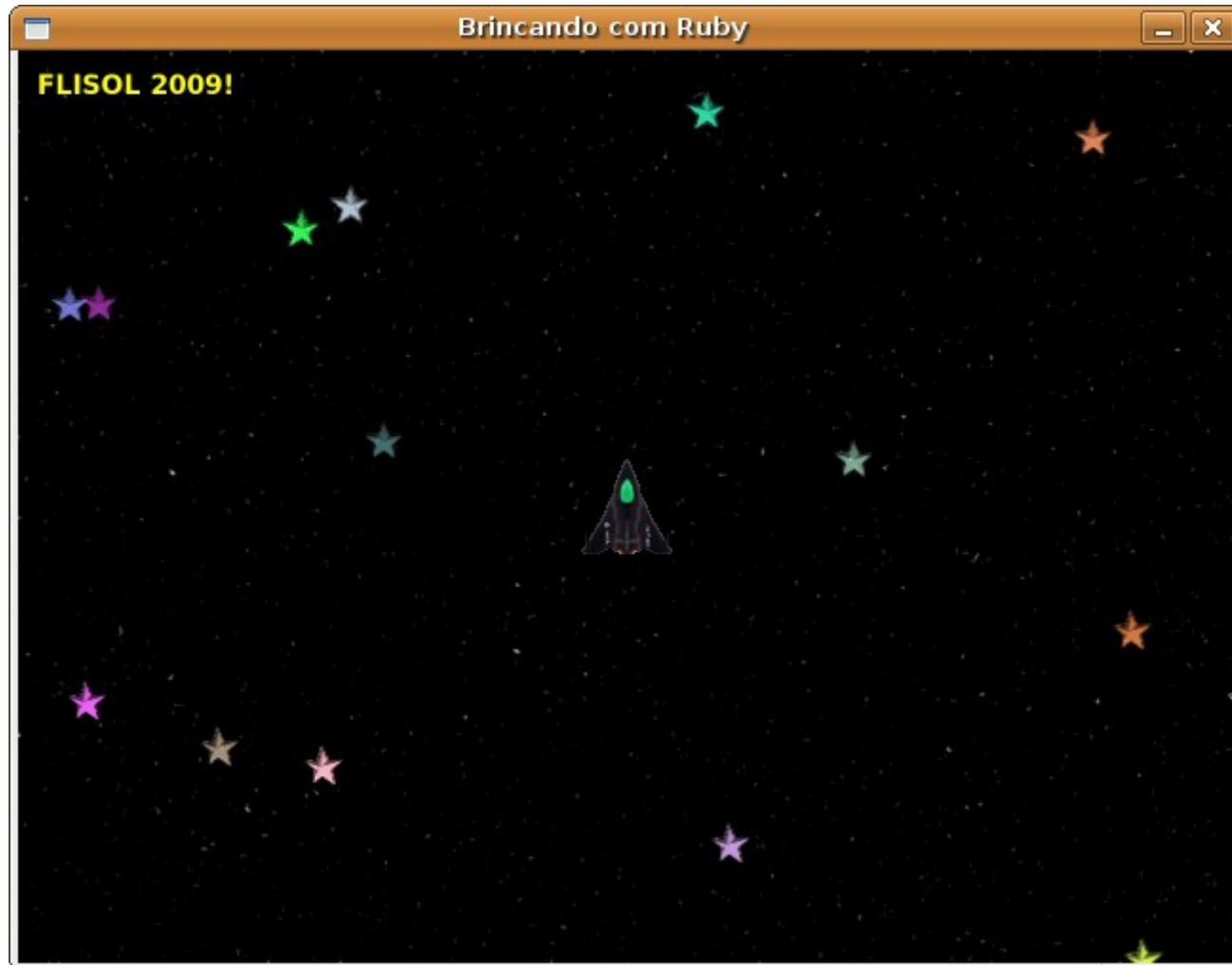
Colisões



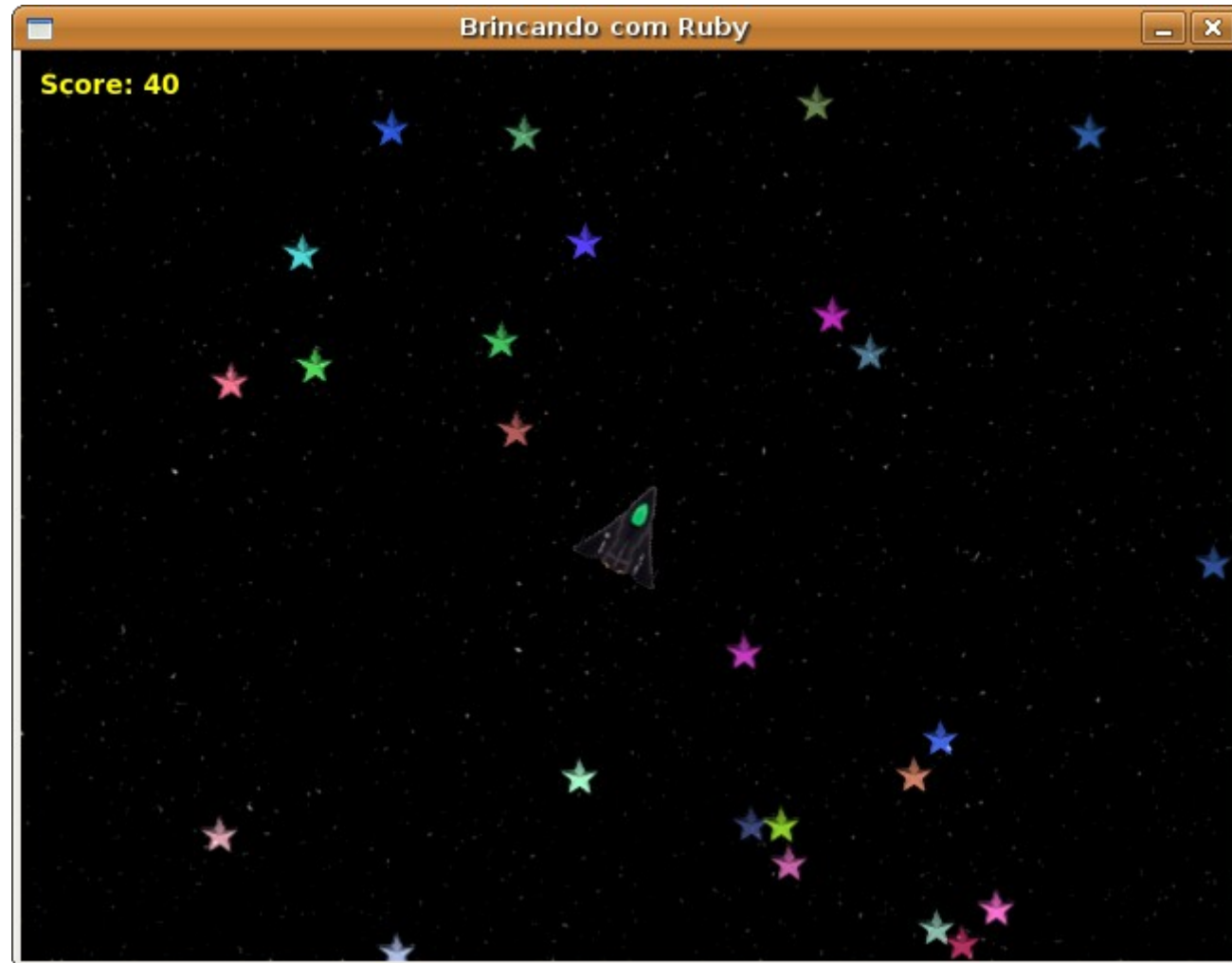
Escrevendo



Escrevendo



Pontuação



Som



Música



Obrigado!

Dúvidas?



Contato

Vítor Baptista
vitor@vitorbaptista.com
<http://vitorbaptista.com>

