CAPÍTULO 5

Metaprogramação

"A ciência nunca resolve um problema sem criar mais dez"
— George Bernard Shaw

5.1 – MÉTODOS DE CLASSE

Classes em Ruby **também são objetos**:

```
Pessoa.class
# => Class

c = Class.new
instancia = c.new
```

Variáveis com letra maiúscula representam constantes em Ruby, que até podem ser modificadas, mas o interpretador gera um *warning*. Portanto, `Pessoa` é apenas uma constante que aponta para um objeto do tipo `Class`.

Se classes são objetos, podemos definir métodos de classe como em qualquer outro objeto:

```
class Pessoa
  # ...
end

def Pessoa.pessoas_no_mundo
  100
end

Pessoa.pessoas_no_mundo
# => 100
```

Há um *idiomismo* para definir os métodos de classe dentro da própria definição da classe, onde `self` aponta para o próprio objeto classe.

```
class Pessoa
  def self.pessoas_no_mundo
```

```
    100
  end

  # ...
end
```

5.2 – Para saber mais: Singleton Classes

A definição **class << object** define as chamadas singleton classes em ruby. Por exemplo, uma classe normal em ruby poderia ser:

```
class Pessoa
  def fala
    puts 'oi'
  end
end
```

Podemos instanciar e invocar o método normalmente:

```
p = Pessoa.new
p.fala   # imprime 'oi'
```

Entretanto, também é possível definir métodos apenas para esse objeto "p", pois tudo em ruby, até mesmo as classes, são objetos, fazendo:

```
def p.anda
  puts 'andando'
end
```

O método "anda" é chamado de singleton method do objeto "p".

Um singleton method "vive" em uma singleton class. Todo objeto em ruby possui 2 classes:

- a classe a qual foi instanciado
- sua singleton class

A singleton class é exclusiva para guardar os métodos desse objeto, sem compartilhar com outras instâncias da mesma classe.

Existe uma notação especial para definir uma singleton class:

```
class << Pessoa
  def anda
    puts 'andando'
  end
end
```

Definindo o código dessa forma temos o mesmo que no exemplo anterior, porém definindo o método anda explicitamente na singleton class.É possível ainda definir tudo na mesma classe:

```
class Pessoa
  class << self
    def anda
      puts 'andando'
    end
  end
end
```

Mais uma vez o método foi definido apenas para um objeto, no caso , o objeto "Pessoa", podendo ser executado com:

```
Pessoa.anda
```


5.3 – Exercícios – Ruby Object Model

1. Crie um novo arquivo chamado **metaprogramacao.rb**. Independente da instância seria interessante saber o número total de restaurantes criados. Na mesma classe restaurante, crie uma variável de classe chamada **total**.

```
class Restaurante
  def initialize(nome)
    puts "criando um novo restaurante: #{nome}"
    @@total ||= 0
    @@total = @@total + 1
    puts "Restaurantes criados: #{@@total}"
    @nome = nome
  end
end
```

Execute novamente e analise o resultado.

2. Crie um método **self** chamado **relatorio** que mostra a quantidade total de restaurantes instanciados. (Opcional: Com esse método, você pode limpar um pouco o initialize).

```
def self.relatorio
  puts "Foram criados #{@@total} restaurantes"
end

# Faça mais uma chamada
Restaurante.relatorio
```

> **Método de Classe é diferente de static do Java**
>
> Se você trabalha com **java** pode confundir o **self** com o **static**. Cuidado! O método definido como self roda apenas na classe, não funciona nas instâncias. Você pode testar fazendo:
>
> ```
> Restaurante.relatorio
> restaurante_um.relatorio
> ```
>
> A invocação na instância dará um: NoMethodError: undefined method 'relatorio' for #<Restaurante:0x100137b48 @nome="Fasano", @nota=10>

3. Caso sua classe possua muitos métodos de classe, é recomendado agrupá-los. Muitos códigos ruby utilizaram essa técnica, inclusive no próprio Rails Sua classe ficará então assim:

```
class Restaurante
   # initialize, qualifica ...

  class << self
    def relatorio
      puts "Foram criados #{@@total} restaurantes"
    end
  end

end
```

Execute e veja que o comportamento é o mesmo.

5.4 - Convenções

Métodos que retornam booleanos costumam terminar com ?, para que pareçam perguntas aos objetos:

```
texto = "nao sou vazio"
texto.empty? # => false
```

Já vimos esta convenção no método `respond_to?`.

Métodos que tem efeito colateral (alteram o estado do objeto, ou que costumem lançar exceções) geralmente terminam com `!` (bang):

```
conta.cancela!
```

A comparação entre objetos é feita através do método `==` (sim, é um método!). A versão original do método apenas verifica se as referências são iguais, ou seja, se apontam para os mesmos objetos. Podemos reescrever este comportamento e dizer como comparar dois objetos:

```
class Pessoa
  def ==(outra)
    self.cpf == outra.cpf
  end
end
```

Na definição de métodos, procure sempre usar os parênteses. Para a chamada de métodos, não há convenção. Prefira o que for mais legível.

Nomes de variável e métodos em Ruby são sempre minúsculos e separados por `'_'` (underscore). Variáveis com nomes maiúsculo são sempre constantes. Para nomes de classes, utilize as regras de **CamelCase**, afinal nomes de classes são apenas constantes.

5.5 – Polimorfismo

Ruby também tem suporte a herança simples de classes:

```
class Animal
  def come
    "comendo"
  end
end

class Pato < Animal
  def quack
    "Quack!"
  end
end

pato = Pato.new
pato.come # => "comendo"
```

Classes filhas herdam todos os métodos definidos na classe mãe.

A tipagem em Ruby não é explícita, por isso não precisamos declarar quais são os tipos dos atributos. Veja este exemplo:

```
class PatoNormal
  def faz_quack
    "Quack!"
  end
end

class PatoEstranho
  def faz_quack
    "Queck!"
  end
end

class CriadorDePatos
  def castiga(pato)
    pato.faz_quack
  end
end

pato1 = PatoNormal.new
pato2 = PatoEstranho.new
c = CriadorDePatos.new
c.castiga(pato1) # => "Quack!"
c.castiga(pato2) # => "Queck!"
```

Para o criador de patos, não interessa que objeto será passado como parâmetro. Para ele basta que o objeto saiba fazer *quack*. Esta característica da linguagem Ruby é conhecida como *Duck Typing*.

"If it walks like a duck and quacks like a duck, I would call it a duck."

5.6 - Exercícios - Duck Typing

1. Para fazer alguns testes de herança e polimorfismo abra o diretório **ruby** e crie um novo arquivo chamado **duck_typing.rb**. Nesse arquivo faça o teste do restaurante herdando um método da classe **Franquia**.

```
class Franquia
  def info
    puts "Restaurante faz parte da franquia"
  end
end

class Restaurante < Franquia
end

restaurante = Restaurante.new
restaurante.info
```

Execute o arquivo no terminal e verifique a utilização do método herdado.

2. Podemos em ruby fazer a sobrescrita do método e invocar o método da classe mãe.

```
class Franquia
  def info
    puts "Restaurante faz parte da franquia"
  end
end

class Restaurante < Franquia
  def info
    super
    puts "Restaurante Fasano"
  end
end

restaurante = Restaurante.new
restaurante.info
```

3. Como a linguagem ruby é implicitamente tipada, não temos a garantia de receber um objeto do tipo que esperamos. Isso faz com que acreditamos que o objeto é de um tipo específico caso ele possua um método esperado. Faça o teste abaixo:

```
class Franquia
  def info
    puts "Restaurante faz parte da franquia"
  end
end

class Restaurante < Franquia
  def info
```

```
    super
    puts "Restaurante Fasano"
  end
end

# metodo importante
# recebe franquia e invoca o método info
def informa(franquia)
  franquia.info
end

restaurante = Restaurante.new
informa restaurante
```

5.7 – Modulos

Modulos podem ser usados como namespaces:

```
module Caelum
  module Validadores

    class ValidadorDeCpf
      # ...
    end

    class ValidadorDeRg
      # ...
    end

  end
end

validador = Caelum::Validadores::ValidadorDeCpf.new
```

Ou como **mixins**, conjunto de métodos a ser incluso em outras classes:

```
module Comentavel
  def comentarios
    @comentarios ||= []
  end

  def recebe_comentario(comentario)
    self.comentarios << comentario
  end
end

class Revista
  include Comentavel
  # ...
end

revista = Revista.new
revista.recebe_comentario("muito ruim!")
puts revista.comentarios
```

5.8 - Metaprogramação

Por ser uma linguagem dinâmica, Ruby permite adicionar outros métodos e operações aos objetos em tempo de execução.

Imagine que tenho uma pessoa:

```
pessoa = Object.new()
```

O que aconteceria, se eu tentasse invocar um método inexistente nesse objeto? Por exemplo, se eu tentar executar

```
pessoa.fala()
```

O interpretador retornaria com uma mensagem de erro uma vez que o método não existe.

Mas e se eu desejasse, **em tempo de execução**, adicionar o comportamento (ou método) `fala` para essa pessoa. Para isso, tenho que **definir** que uma **pessoa** possui o método **fala**:

```
pessoa = Object.new()

def pessoa.fala()
  puts "Sei falar"
end
```

Agora que tenho uma pessoa com o método `fala`, posso invocá-lo:

```
pessoa = Object.new()

def pessoa.fala()
  puts "Sei falar"
end

pessoa.fala()
```

Tudo isso é chamado **meta-programação**, um recurso muito comum de linguagens dinâmicas. Meta-programação é a capacidade de gerar/alterar código em tempo de execução. Note que isso é muito diferente de um gerador de código comum, onde geraríamos um código fixo, que deveria ser editado na mão e a aplicação só rodaria esse código posteriormente.

Levando o dinamismo de Ruby ao extremo, podemos criar métodos que definem métodos em outros objetos:

```ruby
class Aluno
  # nao sabe nada
end

class Professor
  def ensina(aluno)
    def aluno.escreve
      "sei escrever!"
    end
  end
end

juca = Aluno.new
juca.respond_to? :escreve
# => false

professor = Professor.new
professor.ensina juca
juca.escreve
# => "sei escrever!"
```

A criação de métodos acessores é uma tarefa muito comum no desenvolvimento orientado a objetos. Os métodos são sempre muito parecidos e os desenvolvedores costumam usar recursos de geração de códigos das IDEs para automatizar esta tarefa.

Já vimos que podemos criar código Ruby que escreve código Ruby (métodos). Aproveitando essa possibilidade do Ruby, existem alguns métodos de classe importantes que servem apenas para criar alguns outros métodos nos seus objetos.

```ruby
class Pessoa
  attr_accessor :nome
end

p = Pessoa.new
p.nome = "Joaquim"
puts p.nome
# => "Joaquim"
```

A chamada do método de classe `attr_acessor`, define os métodos `nome` e `nome=` na classe `Pessoa`.

A técnica de *código gerando código* é conhecida como **metaprogramação**, ou **metaprogramming**, como já definimos.

Outro exemplo interessante de metaprogramação é como definimos a visibilidade dos métodos em Ruby. Por padrão, todos os métodos definidos em uma classe são públicos, ou seja, podem ser chamados por qualquer um.

Não existe nenhuma palavra reservada (*keyword*) da linguagem para mudar a visibilidade. Isto é feito com um método de classe. Toda classe possui os métodos `private`, `public` e `protected`, que são métodos que alteram outros métodos, mudando a sua visibilidade (código alterando código == **metaprogramação**).

Como visto, por padrão todos os métodos são públicos. O método de classe `private` altera a visibilidade de todos os métodos definidos após ter sido chamado:

```
class Pessoa

  private

  def vai_ao_banheiro
    # ...
  end
end
```

Todos os métodos após a chamada de `private` são privados. Isso pode lembrar um pouco C++, que define regiões de visibilidade dentro de uma classe (seção pública, privada, ...). Um método privado em Ruby **só pode ser chamado em self** e o `self` deve ser **implícito**. Em outras palavras, não podemos colocar o `self` explicitamente para métodos privados, como em `self.vai_ao_banheiro`.

Caso seja necessário, o método `public` faz com que os métodos em seguida voltem a ser públicos:

```
class Pessoa

  private
  def vai_ao_banheiro
    # ...
  end

  public
  def sou_um_metodo_publico
    # ...
  end
end
```

O último modificador de visibilidade é o `protected`. Métodos protected só podem ser chamados em self (implícito ou explícito). Por isso, o `protected` do Ruby acaba sendo semelhante ao protected do Java e C++, que permitem a chamada do método na própria classe e em classes filhas.

5.9 - Exercícios - Metaprogramação

1. Alguns restaurantes poderão receber um método a mais chamado "cadastrar_vips". Para isso precisaremos abrir em tempo de execução a classe Restaurante.

```
# faça a chamada e verifique a exception NoMethodError
restaurante_um.cadastrar_vips

# adicione esse método na classe Franquia
def expandir(restaurante)
  def restaurante.cadastrar_vips
    puts "Restaurante #{self.nome} agora com área VIP!"
  end
end

# faça a franquia abrir a classe e adicionar o método
franquia.expandir restaurante_um
restaurante_um.cadastrar_vips
```

2. Um método útil para nossa franquia seria a verificação de nome de restaurantes cadastrados. vamos fazer isso usando **method_missing**.

```
# Adicione na classe Franquia
def method_missing(name, *args)
  @restaurantes.each do |r|
    return "O restaurante #{r.nome} já foi cadastrado!"
        if r.nome.eql? *args
  end
  return "O restaurante #{args[0]} não foi cadastrado ainda."
end

# Faça as chamadas e analise os resultados
puts franquia.já_cadastrado?("Fasano")
puts franquia.já_cadastrado?("Boteco")
```

CAPÍTULO ANTERIOR:

[Mais Ruby: classes, objetos e métodos]

PRÓXIMO CAPÍTULO:

[Ruby on Rails]

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter