11

Conceito de Classes e métodos.

Estruturas e classes: uma introdução a Orientação a Objetos

A bagunça dos defs

Olhemos nossa função soma, que soma dois vetores para calcular uma nova posição:

```
def soma(vetor1, vetor2)
    [vetor1[0] + vetor2[0], vetor1[1] + vetor2[1]]
end
```

Por mais que ela seja uma função que faz sentido, o que ela tem em comum com a função `copia_mapa` ?

```
def copia_mapa(mapa)
    mapa.join("\n").tr("F", " ").split("\n")
end
```

São lógica de negócio, mas mais nada. Seguindo somente esse critério de separação de lógica de negócios, é fácil imaginar que um projeto ou jogo grande vai ficar com "infinitas" funções juntas. Mais ainda, quem sabe como somar um movimento a uma posição é o gerenciamento de posições. Quem sabe copiar um mapa é o próprio mapa. Nesse instante todas essas responsabilidades estão no mesmo lugar, jogadas em um único arquivo. Até mesmo o herói... como assim o herói é um array de duas posições? O herói é um herói, é você, sou eu... ele não é um mero array.

Apesar do nosso foco ser uma introdução a computação e programação, por utilizarmos a linguagem Ruby como exemplo, desenvolveremos um pouco mais nosso jogo no sentido de fazê-lo suportar o mínimo de orientação a objetos... e começaremos pelo que há de mais importante para nós, o jogador, nosso herói.

Extraindo uma primeira estrutura

Até agora representamos um herói com um array de tamanho dois, o que é algo muito estranho. Pensando em valores, um herói é um herói, não um array. E para movê-la basta mudar sua linha e coluna, mandar se movimentar. Isto é, seria interessante ser possível criar um único valor, `Heroi` , que tivesse dois valores dentro dele, a `linha` e a `coluna` atual.

Queremos criar uma estrutura que representa um herói, uma abstração de nosso herói. Essa abstração não é nosso herói de verdade, somente uma definição de como um herói se comporta no mundo real, criamos então uma classe:

```
class Heroi
end
```

Agora podemos criar um herói:

```
class Heroi
end

heroi = Heroi.new
```

Mas uma herói sem linha nem coluna não tem graça. Queremos dizer que um herói tem dois atributos, sua linha e sua coluna:

```
class Heroi
    attr_accessor :linha, :coluna
end
heroi = Heroi.new

heroi.linha = 3
heroi.coluna = 6

puts heroi.linha # 3
puts heroi.coluna # 6
```

Isto é, teremos uma variável chamada `heroi` que tem um valor. Esse valor é a referência para um objeto, uma instância de `Heroi`, sendo que dentro desse objeto teremos os valores `linha` e `coluna`, no nosso caso do herói esses valores são `3` e `6`.

Da mesma maneira que criamos um herói, um objeto do tipo herói, podemos criar dois. Cada um tem valores diferentes, pois tem espaços diferentes na memória:

```
class Heroi
    attr_accessor :linha, :coluna
end

guilherme = Heroi.new
guilherme.linha = 3
guilherme.coluna = 6

paulo = Heroi.new
paulo.linha = 5
paulo.coluna = 3

puts guilherme.linha # 3
puts guilherme.coluna # 6

puts paulo.linha # 5
puts paulo.coluna # 3
```

O que fizemos até agora? Definimos uma abstração, como funciona um v, uma classe `Heroi`. Ela dá as regras de como um herói pode ser criado (instanciado), quais seus atributos (variáveis de instância), comportamentos (funções) entre outros. Para nós já é ótimo saber que nosso herói no sistema agora é realmente um herói, e não uma gambiarra de um vetor. Portanto o código final do arquivo `heroi.rb` é:

```
class Heroi
    attr_accessor :linha, :coluna
end
```

Usando uma estrutura

Devemos usar nosso jogador como uma instância de `Heroi`, para isso alteraremos o momento em que ele é encontrado, o `encontra_jogador`:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
            return [linha, coluna_do_heroi]
        end
    end
    nil
end
```

Quando encontramos um jogador, instanciamos, criamos um novo objeto do tipo `Heroi` na linha e coluna adequada:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
            heroi = Heroi.new
            heroi.linha = linha
            heroi.coluna = coluna_do_heroi
            return heroi
        end
    end
    nil
end[/cpde]

Já que usamos a classe %%Heroi%% precisamos carregá-la:

[code ruby]
require_relative 'ui'
require_relative 'heroi'

# ...
```

Tentamos encontrar o jogador na função `joga`:

```
heroi = encontra_jogador mapa
```

```
nova_posicao = calcula_nova_posicao heroi, direcao
if !posicao_valida? mapa, nova_posicao
    next
end

mapa[heroi[0]][heroi[1]] = " "
mapa[nova_posicao[0]][nova_posicao[1]] = "H"
```

Primeiro usamos ela para encontrar a posição do herói no mapa, o que podemos mudar para:

```
mapa[heroi.linha][heroi.coluna] = " "
```

O herói também é passado como argumento para o `calcula_nova_posicao` :

```
nova_posicao = calcula_nova_posicao heroi, direcao
```

Que brinca com os valores `[0]` e `[1]` de nosso antigo array.

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    movimentos = {
        "W" => [-1, 0],
        "S" => [+1, 0],
        "A" => [0, -1],
        "D" => [0, +1]
        }
    movimento = movimentos[direcao]
    heroi[0] += movimento[0]
    heroi[1] += movimento[1]
    heroi
end
```

Isso não faz mais sentido, afinal um herói não tem mais esses acessos, ele possui uma linha e uma coluna, portanto:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    movimentos = {
        "W" => [-1, 0],
        "S" => [+1, 0],
        "A" => [0, -1],
        "D" => [0, +1]
        }
    movimento = movimentos[direcao]
    heroi.linha += movimento[0]
    heroi.coluna += movimento[1]
    heroi
end
```

Code smell: feature envy

Opa. Nosso `calcula_nova_posicao` tem algo de estranho. Das 6 instruções contidas no código, 4 delas (dois terços!) envolvem nosso herói! O código nos indica que a função `calcula_nova_posicao` está com inveja do herói, ela queria ter as funcionalidades dele e por isso fica perguntando cinquenta coisas pra ele: `dup` por favor, `linha` por favor, `coluna` por favor, `retorna você` por favor. Quanta inveja, inveja de funcionalidade, ::feature envy::.

Essa característica é um fedor de código que indica que essa função poderia pertencer a quem ela tanto "gosta". Afinal, perguntemos a nós mesmos, quem entende mais de um personagem heróico do que ele mesmo? Quem sabe movimentar um herói? Ele mesmo! Além disso, não faz nenhum sentido a função `calcula_nova_posicao` ser invocada se um herói não existe. Por todos esses motivos podemos mover a função para dentro de nosso `Heroi` :

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(heroi, direcao)
        heroi = heroi.dup
        movimentos = {
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
        heroi.coluna += movimento[1]
        heroi
    end
end
```

Mas se a função já está dentro do próprio herói, não precisamos receber ele como argumento. Removemos então o argumento:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        # ...
    end
end
```

Já que estamos rodando esse código dentro de nosso herói, e não temos mais ele como argumento, a variável `heroi` não existe mais para executarmos o `dup` . Como invocar então o método `dup` no próprio objeto? Como dizer pro código de um método dentro de um objeto para invocar um outro método nele mesmo, nele próprio? Usamos a palavra próprio, em inglês ::self:::

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        heroi = self.dup
        movimentos = {
```

```
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
        heroi.coluna += movimento[1]
        heroi
    end
end
```

Por fim, quando invocamos um método em nós mesmos, o uso do `self` é na verdade opcional, portanto podemos:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        heroi = dup
        movimentos = {
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
        heroi.coluna += movimento[1]
        heroi
    end
end
```

Nossa função `joga` deve agora invocar o método no próprio objeto:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = heroi.calcula_nova_posicao direcao
        if !posicao_valida? mapa, nova_posicao
            next
        end

        mapa[heroi.linha][heroi.coluna] = " "
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"

        mapa = move_fantasmas mapa
        if jogador_perdeu?(mapa)
            game_over
            break
```

```
        end
    end
end
```

Boa prática: Buscando quem invoca antes de refatorar

Agora nossa `nova_posicao` também é um objeto do tipo `Heroi`. Ela é usada em nossa função também ao substituirmos o valor dela no mapa, o que já sabemos alterar para utilizar os atributos `linha` e `posicao`:

```
mapa[nova_posicao.linha][nova_posicao.coluna] = "H"
```

Mas ela também é utilizada para validar se a nova posição é válida:

```
if !posicao_valida? mapa, nova_posicao
    next
end
```

Antes de refatorarmos o código de uma função ::sempre:: devemos olhar quem a invoca. Nesse caso, tanto aqui quanto no momento de movimentar um fantasma invocamos a função `posicao_valida?`. Isso significa que se alterarmos o código dela para suportar um `Heroi` ao invés de um array de duas posições, quebraremos o código do fantasma, e não queremos isso nesse instante de nossa refatoração. O que queremos é fazer essa invocação funcionar.

Como então transformar um `Heroi` em um array de duas posições para poder continuar invocando um método que funciona tanto para o herói quanto para o fantasma?

Podemos criar um método novo, seguindo o padrão `to_i` do Ruby, algo como `to_array` (ao invés de `to_a` que é ambíguo demais):

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(heroi, direcao)
        # ...
    end

    def to_array
        [linha, coluna]
    end
end
```

Terminamos então com nosso herói:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        heroi = dup
```

```
        movimentos = {
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
        heroi.coluna += movimento[1]
        heroi
    end

    def to_array
        [linha, coluna]
    end
end
```

E invocamos nosso método `to_array` ao verificar se a nova posição é válida, ficando com o método `joga` :

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = heroi.calcula_nova_posicao direcao
        if !posicao_valida? mapa, nova_posicao.to_array
            next
        end

        mapa[heroi.linha][heroi.coluna] = " "
        mapa[nova_posicao.linha][nova_posicao.coluna] = "H"

        mapa = move_fantasmas mapa
        if jogador_perdeu?(mapa)
            game_over
            break
        end
    end
end
```

Pronto. Nosso jogo já está funcionando. Agora com nosso herói bem representado, e parte do código que o pertence (`calcula_nova_posicao`) dentro dele.

Boa prática: Tell, don't ask

Novamente temos repetição de código. Dado um heróis, fazemos:

```
mapa[heroi.linha][heroi.coluna] = " "
```

Depois, dada uma nova posição, marcamos com `H`:

```
mapa[nova_posicao.linha][nova_posicao.coluna] = "H"
```

Ao invés de pedir informação (::ask::), parece fazer mais sentido mandar executar algo (::tell::):

```
heroi.remove_do mapa
nova_posicao.coloca_no mapa
```

O padrão que estamos adotando é o de mandar executar algo, ao invés de perguntar... ::tell, don't ask::. Dessa maneira, não nos importa se o herói é implementado com `linha` e `coluna`, `x` e `y`, array de tamanho dois. Não importa. Importa que uma posição é capaz de se remover de um mapa. E que ela é capaz de se colocar no mapa.

Definimos então o método `remove_do`, o ato de fazer um ::extract method:::

```
def remove_do(mapa)
    mapa[linha][coluna] = " "
end
```

E o `coloca_no`:

```
def coloca_no(mapa)
    mapa[linha][coluna] = "H"
end
```

Atributos e attr_accessor

O ::attr_accessor:: cria na verdade dois métodos. Um para alterar e outro para ler o valor de um atributo. Como ler um atributo diretamente de dentro de uma classe? Através do uso de um `@`, como no caso:

```
def coloca_no(mapa)
    mapa[@linha][@coluna] = "H"
end
```

Como estamos utilizando um `attr_accessor`, tanto faz mantermos o `@` ou não, portanto manteremos sem o `@`.

Além do `attr_accessor` existem também o `attr_reader`, que fornece somente um leitor, e o `attr_writer`, que disponibiliza somente o método de escrita.

Estrutura ou Classe?

Costumamos chamar de ::estrutura:: a definição de um conjunto de valores agrupados. No começo de nosso código utilizamos um `Heroi` para somente agrupar valores, nada mais. Ao criarmos um valor de uma estrutura costumamos chamar esse valor, essa instância, de dados da estrutura.

Uma classe é a definição de um agrupamento tanto de valores como de comportamentos, funções. Nesse caso chamamos essas funções de ::métodos::, como vimos antes. Ao criarmos uma instância de uma classe chamamos esse valor de objeto.

Já havíamos utilizados objetos antes. Na realidade todo valor em Ruby é um objeto. As ::String::s, ::Array::s, ::Fixnum:: e ::Float:: são todos tipos (classes) que foram criadas instâncias. Por exemplo, ao usarmos o valor `15` estamos com uma instância de `Fixnum` cujo valor interno é `15` . Ao criarmos um array de números temos na verdade um objeto (o array) que aponta para diversos objetos (cada um dos números).

Dependendo da linguagem, a definição varia um pouco, não se assuste ao aprender uma nova linguagem. Uma funcionalidade de uma pode ter um nome um pouco diferente em outra.

Code smell: classes anêmicas

Aqui em Ruby acabamos por usar basicamente classes como classes de verdade, e não como meras estruturas bobas (comumente chamadas de classes anêmicas).

A verdade por trás de métodos, funções e lambdas

Já que uma função é um nome para um trecho de código, podemos pensar que uma função também poderia ser um tipo de variável, assim como um array é. Na prática é isso que acontece! Por trás dos panos, tanto uma função quanto um valor qualquer é referenciado através de um nome, o nome da função ou da variável.

Só que em Ruby não conseguimos fazer diretamente uma variável referenciar uma função que acaba de ser definida como vimos até agora devido a regra do parenteses. Em Javascript isso seria possível:

```
function bemvindo(nome) {
    println("Bem vindo " + nome + "!");
}

var minhaFuncao = bemvindo;
minhaFuncao("Guilherme");
```

Em Ruby, o código análogo invocaria a função, sem passar parâmetros, ao tentar referência-la:

```
def bemvindo(nome)
  puts "Bem vindo " + nome + "!"
end

minhaFuncao = bemvindo # tenta invocar a função sem parâmetro!
minhaFuncao "Guilherme" # não faz sentido nenhum
```

Para fazer isso, Ruby permite a criação de funções "soltas", literalmente uma função, chamadas nesse contexto de ::lambda literal::, que não são métodos através de uma `Proc` , uma função que não está atrelada a nenhum objeto:

```
bemvindo = -> (nome) {
  puts "Bem vindo " + nome + "!"
}
```

Podemos agora invocá-la, com uma sintaxe bem feia - vendo por outro lado, mais educada - deixando claro que estamos invocando (::call::) algo:

```
bemvindo.("Guilherme") # Bem vindo Guilherme!
bemvindo.call("Guilherme") # Bem vindo Guilherme!
```

Ou ainda reatribuí-la a outra variável:

```
minhaFuncao = bemvindo
minhaFuncao.("Guilherme") # Bem vindo Guilherme!
```

Como o método é a invocação de uma função no objeto, temos em Ruby que sempre estamos invocando métodos. O conceito de função pura não faz sentido, uma vez que até mesmo o ::lambda literal:: é um objeto onde chamamos um método, o método `call`. Portanto não se preocupe, isso não significa que Ruby é ou não é funcional, somente que todo valor em Ruby é um objeto e que as funções que invocamos são chamadas de métodos.

Como vício de linguagem, acabamos usando a palavra ::função:: para dizer que estamos invocando ou definindo esses métodos, afinal uma função está mais próximo da definição formal disso que estamos invocando. Novamente, não se preocupe, ao conversar com seus colegas use o vocabulário que deixe claro o que é que está fazendo.

Em Ruby, definimos e invocamos métodos - mas se você chamá-los de função, ninguém sai machucado.

gets nao responde por isso deixei a palavra funcao.

Resumindo

Vimos como criar uma estrutura que basicamente armazena dados, mas em Ruby ela já é uma classe, com métodos e atributos. Aprendemos a utilizá-la, instanciando um objeto a partir da definição descrita de nossa classe.

Ao notar que nosso código utilizava muitos valores da estrutura movemos o comportamento para dentro dela, criando nosso primeiro método. Aprendemos o que significa invocar um método para mandar fazer algo, ao invés de perguntar por alguma informação, favorecendo o encapsulamento da lógica por trás do comportamento, incluindo seus atributos. Vimos como expor, acessando e escrevendo valores em atributos.

Outras linguagens podem definir os dois tipos bem distintos: estruturas que costumam somente agrupar valores, e classes que agrupam valores e comportamentos.

Por fim, vimos que todo valor em Ruby é de alguma forma um objeto portanto sempre que utilizamos o conceito de funções estavámos falando na verdade de métodos - conceitualmente não há nada de errado em chamá-las de funções, mas em Ruby tais funções são também métodos. Por fim, vimos que se um número e uma `String` podem ser valores que damos nomes (aplicamos a variáveis), funcões também podem ser tratadas assim através de lambdas.