08

Começando o jogo com boas práticas de programação

Jogo da forca

Nosso próximo grande jogo é o da forca, onde o jogador deve adivinhar uma palavra, chutando cada vez uma nova letra ou a palavra inteira.

Começamos nosso arquivo `forca.rb` com a mensagem de boas vindas de acordo com o nome do jogador:

```
def da_boas_vindas
    puts "Bem vindo ao jogo da forca"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end
```

O próximo passo é a escolha de uma palavra secreta. Queremos mostrar o número de letras contidas na palavra secreta, para isso usamos o método `size` :

```
def sorteia_palavra_secreta
    puts "Escolhendo uma palavra..."
    palavra_secreta = "programador"
    puts "Escolhida uma palavra com #{palavra_secreta.size} letras... boa sorte!"
    palavra_secreta
end
```

Definimos também a função que pergunta se o jogador deseja jogar novamente:

```
def nao_quer_jogar?
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    nao_quero_jogar = quero_jogar.upcase == "N"
end
```

O nosso laço principal jogará o jogo diversas vezes:

```
nome = da_boas_vindas

loop do
    joga nome
    break if nao_quer_jogar?
end
```

Agora a única grande diferença está em nosso `joga`. Nossa função de jogar deve escolher a palavra secreta, configurar 0 erros até agora e começar com zero pontos.

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    # laco principal a escrever

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

O laço principal deve ser executado até o número de erros chegar a 5, isto é, enquanto `erros` for menor que `5`:

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        # o jogo
    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

Agora definimos a função que lê a palavra chutada. Mas diferentemente de antes, não temos um limite de tentativas e não armazenamos a tentativa atual, somente os chutes e o número de erros:

```
def pede_um_chute(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Podemos continuar com nosso laço principal, pedindo o chute e guardando ele no array de chutes:

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta
```

```
    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, erros
        chutes << chute

        # colocar as regras de acerto e erro aqui!

    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

Chute de uma palavra completa e a comparação com ==

O jogador ganha 100 pontos a cada palavra chutada certa e perde 30 pontos a cada palavra inteira chutada errada. Ele não perde nem ganha pontos ao chutar uma letra, afinal com cinco chutes errados ele perde o jogo.

Falta então colocarmos as regras de acerto e erro. Primeiro devemos conferir se o usuário chutou uma única letra, isto é, se sua entrada tem tamanho 1, ou se o chute foi da palavra inteira:

```
if chute.size == 1
    # chutou uma única letra
else
    # chutou a palavra inteira
end
```

Mas já falamos qual o problema de comentários. Vamos então extrair uma variável, algo que nos diga se ele `chutou_uma_unica_letra`:

```
chutou_uma_unica_letra = chute.size == 1
if chutou_uma_unica_letra

else

end
```

Caso ele tenha chutado a palavra inteira, e acertado, mostramos a mensagem de parabéns e paramos a jogada com o `break` que já conhecemos:

```
chutou_uma_unica_letra = chute.size == 1
if chutou_uma_unica_letra

else
    acertou = chute == palavra_secreta
```

```
    if acertou
        puts "Parabéns! Acertou!"
        pontos_ate_agora += 100
        break
    else
    end
end
```

Caso ele erre, tiramos os `30` pontos e aumentamos um erro:

```
chutou_uma_unica_letra = chute.size == 1
if chutou_uma_unica_letra

else
    acertou = chute == palavra_secreta
    if acertou
        puts "Parabéns! Acertou!"
        pontos_ate_agora += 100
        break
    else
        puts "Que pena... errou!"
        pontos_ate_agora -= 30
        erros += 1
    end
end
```

Nosso jogo já funciona para chutes inteiros da palavra, por exemplo se eu tentar chutar a palavra *programadores* (-30 pontos), e depois a palavra *programador* (+100 pontos), o jogo para com meus 70 pontos:

```
Erros até agora: 0
Chutes até agora: []
Entre com a letra ou palavra
programadores
Será que acertou? Você chutou programadores
Que pena... errou!
...
Erros até agora: 1
Chutes até agora: ["programadores"]
Entre com a letra ou palavra
programador
Será que acertou? Você chutou programador
Parabéns! Acertou!
Você ganhou 70 pontos.
...
```

Encontrando um algoritmo

Mas precisamos agora suportar os chutes de letras. Cada vez que o jogador tenta uma única letra devemos verificar se a nossa `String palavra_secreta` inclue o caracter escolhido. Seria interessante dizer quantas vezes encontramos tal letra. Se queremos contar quantas vezes encontramos uma letra em uma palavra, como podemos fazer? Vamos pensar

como fazemos isso como crianças. Esse é o segredo para escrever em código a solução de qualquer problema lógico: lembre-se como você resolve o problema, e ai implemente ele. Mas cuidado, lembre-se como uma criança, pois a criança usa as estruturas mais básicas de lógica (ela não usa potência, nem log nem coisas complexas, somente ifs e fors).

Por exemplo, dada a palavra mágica, `programador` para uma criança, como ela conta quantas vezes aparece a letra `o`? Ela escreve zero em um cantinho do papel (ou na cabeça, na memória!). Ai ela olha a primeira letra da palavra (aponta para ela com o dedo!). Essa letra é a letra `o` que estou procurando? Não. Vai para a próxima. É? Não. Próxima. É? Sim! Então soma um no cantinho do papel (na memória). Continua até o fim.

No fim, se o número é zero, significa que a letra não está lá dentro, aumenta um erro. Se o número é diferente de zero, ele representa quantas vezes encontrou. Pronto. Temos nossa estrutura lógica para resolver o problema, temos nosso algoritmo.

Estou usando o exemplo de uma criança, mas não se assuste, mesmo que vá resolver um problema de grafos complexos em que deseja encontrar a menor e mais barata rota entre três cidades via avião a idéia é ter em mente que descrever o problema em português e como você (um adulto, uma criança grande, não importa) resolveria ele com estruturas básicas de lógica é a chave para escrever o algoritmo na linguagem de programação que deseja. Com o passar do tempo, claro, você começa a eliminar esse passo. Não tenha pressa nenhuma. Dois meses, seis meses, seis anos. Os programadores que já viu escrevendo código tem anos de experiência e por isso já fizeram o mesmo que você está fazendo milhares de vezes. Com milhares de pontos de XP você também será capaz de escrever o algoritmo de contar letras de cabeça.

Implementando o algoritmo

Tomemos a estrutura - *o algoritmo* - ao pé da letra, como foi descrita aqui:

```
dada a palavra mágica, `programador` para uma criança,
Ela escreve zero em um cantinho do papel (ou na cabeça, na memória!).
Ai ela olha a primeira letra da palavra (aponta para ela com o dedo!).
Essa letra é a letra `o` que estou procurando? Não.
Vai para a próxima.
É? Não.
Próxima.
É? Sim!
Então soma um no cantinho do papel (na memória).
Continua até o fim.

No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
se o número é diferente de zero,
ele representa quantas vezes encontrou.
```

Agora temos que traduzir nossa estrutura para Ruby. Sem problemas. Comecemos com as variáveis:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0
```

```
Ai ela olha a primeira letra da palavra (aponta para ela com o dedo!).
Essa letra é a letra `o` que estou procurando? Não.
Vai para a próxima.
É? Não.
Próxima.
É? Sim!
Então soma um no cantinho do papel (na memória).
Continua até o fim.

No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
se o número é diferente de zero,
ele representa quantas vezes encontrou.
```

Agora precisamos traduzir nosso laço, quando a criança passa para cada letra:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0

for i = 0..(palavra_secreta.size - 1)
    letra = palavra_secreta[i]
    Essa letra é a letra `o` que estou procurando? Não.
    É? Não.
    É? Sim!
    Então soma um no cantinho do papel (na memória).
end

No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
se o número é diferente de zero,
ele representa quantas vezes encontrou.
```

Se a letra for a letra correta, devemos atualizar nosso contador:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0

for i = 0..(palavra_secreta.size - 1)
    letra = palavra_secreta[i]
    if letra == chute
        total_encontrado += 1
    end
end

No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
```

```
se o número é diferente de zero,
ele representa quantas vezes encontrou.
```

E no fim mostrar o resultado:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0

for i = 0..(palavra_secreta.size - 1)
    letra = palavra_secreta[i]
    if letra == chute
        total_encontrado += 1
    end
end

if total_encontrado == 0
    puts "Letra não encontrada!"
    erros += 1
else
    puts "Letra encontrada #{total_encontrado} vezes!"
end
```

Claro, no nosso código já temos a variável `palavra_secreta` e `chute`. Portanto nossa função `joga` fica:

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, erros
        chutes << chute

        chutou_uma_unica_letra = chute.size == 1
        if chutou_uma_unica_letra
            total_encontrado = 0

            for i = 0..(palavra_secreta.size - 1)
                letra = palavra_secreta[i]
                if letra == chute
                    total_encontrado += 1
                end
            end

            if total_encontrado == 0
                puts "Letra não encontrada!"
                erros += 1
            else
                puts "Letra encontrada #{total_encontrado} vezes!"
            end
```

```
            else
                acertou = chute == palavra_secreta
                if acertou
                    puts "Parabéns! Acertou!"
                    pontos_ate_agora += 100
                    break
                else
                    puts "Que pena... errou!"
                    pontos_ate_agora -= 30
                end
            end

        end

        puts "Você ganhou #{pontos_ate_agora} pontos."
    end
```

Esse código é uma "bela" de uma função. Um códigozão. Em tamanho, não em qualidade. Pelo contrário, é assustador o tamanho dela, e o que fazemos com código horrível que o Guilherme escreve? Refatoramos.

Boa prática: explorando a documentação

Novamente vamos aplicar o conceito de extrair um trecho de código, uma função nova. Repare que o ato de contar quantas vezes um caracter aparece em uma `String` parece ser algo razoavelmente isolável. Então tira ele daí:

```
def conta(texto, caracter)
    total_encontrado = 0

    for i = 0..texto.size
        letra = texto[i]
        if letra == caracter
            total_encontrado += 1
        end
    end

    total_encontrado
end
```

Mas se olharmos na documentação, `String` s possuem um método chamado `chars` que devolve um array de caracteres, que facilita nosso laço:

```
def conta(texto, caracter)
    total_encontrado = 0

    for letra in texto.chars
        if letra == caracter
            total_encontrado += 1
        end
    end
```

```
    total_encontrado
end
```

Ótimo, o código ficou mais simples. Mas já que estamos olhando a documentação, é importante olharmos com calma, o que é aquele método `count`? Ele conta o número de aparições de um caracter em uma `String` (entre outras utilizações possíveis). Portanto podemos remover completamente nossa nova função! Esse código já existe no Ruby!

Mas Guilherme, porque você me levou a criar toda essa função, com um laço, um contador, para depois refatorar e por fim olhar na documentação e perceber que o código já existia?

Não estamos aqui aprendendo a ser *copy e pasters*. Queremos entender como um programa funciona e o que deve ser feito para se resolver um problema. Sempre que você usar uma biblioteca ou função é importante entender como ela resolve seu problema. Podemos não saber os mínimos detalhes, mas se não entendermos o que acontece por trás, a chance de algum bug acontecer é maior, afinal ela pode fazer algo que não sabemos que ela faz.

Nesse caso específico, é vital para um programador entender como funciona um laço com um contador, esse tipo de estrutura lógica é usado em todo canto quando programamos dando comandos, ordens (programação imperativa) como fizemos até agora. Desde o algoritmo comercial do cálculo do total de uma compra até um algoritmo matemático encontrar os fatores primos de um número qualquer, laços com acumuladores (números ou arrays) aparecem e reaparecem.

Como regra geral, sempre que queremos fazer algo com um valor do tipo `X` (por exemplo `String` ou `Fixnum`), vale *muito, muito, muito* a pena olhar a documentação e ver se já existe um método que faça isso.

Como regra geral, você já deve ter percebido, sempre veremos como as coisas funcionam, para então melhorar nosso código.

Como fica então nossa função `joga`?

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, erros
        chutes << chute

        chutou_uma_unica_letra = chute.size == 1
        if chutou_uma_unica_letra
            total_encontrado = palavra_secreta.count(chute[0])
            if total_encontrado == 0
                puts "Letra não encontrada!"
                erros += 1
            else
                puts "Letra encontrada #{total_encontrado} vezes!"
            end
        else
            acertou = chute == palavra_secreta
            if acertou
```

```
                puts "Parabéns! Acertou!"
                pontos_ate_agora += 100
                break
            else
                puts "Que pena... errou!"
                pontos_ate_agora -= 30
                erros += 1
            end
        end

    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

next... evitando chutes repetidos

Por mais que nosso jogo funcione, tem algo de muito estranho ainda, o jogo permite que eu tente duas vezes a mesma letra ou palavra, algo que não faz sentido. Ele poderia me avisar. Como fazer isso?

Pensando como criança, para saber se eu já falei uma letra ou palavra, devo toda vez que falar uma letra ou uma palavra anotar ela. E depois quando falo uma letra ou palavra nova basta conferir se ela já foi anotada antes.

Já temos as palavras anotadas em um array, o `chutes`. Basta então verificarmos:

```
        chute = pede_um_chute chutes, erros
        # verificar aqui se ja chutei
        chutes << chute
```

Como verificar se um valor já está presente em um array? Como criança, tenho que ir para todos os elementos do array, comparando um a um, até encontrar. Se encontrar um igual, verdadeiro, tem. Se nenhum for igual, não encontrei, não tem. Isso é, um laço com acumulador (ou *early return*), como de costume. Ao invés de implementarmos isso, parece razoável imaginar que essa tarefa seja tão comum que a linguagem nos forneça isso. E ela provê uma função chamada `include?` que nos diz se o array inclue o valor que procuramos.

```
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            puts "Você já chutou #{chute}"
        end
        chutes << chute
```

Tentamos e o que acontece?

```
Entre com a letra ou palavra
p
...
Chutes até agora: ["p"]
Entre com a letra ou palavra
```

```
p
Será que acertou? Você chutou p
Você já chutou p
Letra encontrada 1 vezes!
...
Chutes até agora: ["p", "p"]
Entre com a letra ou palavra
```

Ele não ignorou nossa segunda letra, adicionou no array uma segunda vez, apesar de mostrar a mensagem que já havíamos pedido ela. Claro, no nosso `if` nós somente mostramos a mensagem, precisamos escrever o código que continua nosso laço, ignorando todo o resto do código. Mas como fazer isso?

Precisamos de algum tipo de código que seja capaz de continuar o laço, ir para a próxima rodada, a próxima iteração, sem que ele pare. A palavra `break` quebrava o laço, não é o que queremos. Queremos continuar na próxima rodada, *next* (em diversas linguagens, *continue*).

```
while erros < 5
    chute = pede_um_chute chutes, erros
    if chutes.include? chute
        puts "Você já chutou #{chute}"
        next
    end
    chutes << chute

    ...
end
```

Agora sim, ao repetirmos uma letra, ela não é adicionada no array novamente, nem verificada:

```
...
p
Será que acertou? Você chutou p
Letra encontrada 1 vezes!
...
Erros até agora: 0
Chutes até agora: ["p"]
Entre com a letra ou palavra
p
Será que acertou? Você chutou p
Você já chutou p
...
```

Resumindo

Vimos como utilizar diversas características que aprendemos anteriormente de uma linguagem de programação, como laços (`while` , `for` , `loop`), condicionais (`if`), entrada e saída (`puts` e `gets`), além de usar o `strip` , `arrays` , soma e subtração.

Aprendemos a utilização do `break` e do `next` para quebrar ou continuar um laço, o `count` para contar elementos e onde encontrar a documentação da linguagem para evitar reinventar a roda.

Entendemos também como funciona o coração de um programa: a criação e descrição de um processo que resolve um problema, seu algoritmo. Pensando como crianças somos capazes de quebrar o algoritmo em partes tão pequenas que condicionais e laços podem descrevê-lo, portanto criando nosso código a partir dessa descrição.