Condicionais, laços, programação imperativa.

Operadores de comparação

Vimos que podemos utilizar o `=` para atribuir um valor a algo. Por exemplo, utilizamos `nome = gets` para ler do usuário um texto. A esse algo damos o nome de *variável*. No nosso caso, a variável `nome` recebe o valor que foi lido do jogador. No nosso jogo temos três variáveis: o chute, o número secreto e o nome do jogador. Cada variável possui um valor. O número escolhido está fixo como "175", enquanto tanto o nome do jogador quanto o chute são valores lidos a partir da entrada do usuário.

Como o símbolo `=` foi utilizado pela linguagem para atribuir um valor a uma variável, teremos que adotar algum outro símbolo para a comparação de valores. O `==` é o padrão mais comum adotado pelas linguagens de programação para tal situação. Como vimos antes, o código a seguir imprime "verdadeiro" e depois "falso":

```
puts 175 == 175
puts 175 == 174
```

Mas será que existem outros comparadores? Outro muito comum é o diferente, isto é, não (`!`) igual (`=`), como no exemplo a seguir, que imprime "verdadeiro" duas vezes:

```
puts 175 != 375
puts 175 != 174
```

Outros operadores utilizados comumente em operações matemáticas são os de maior (`>`), menor (`<`), maior ou igual (`>=`) ou menor ou igual (`<=`). Todos os exemplos a seguir imprimem "verdadeiro":

```
puts 175 > 174
puts 174 < 175
puts 170 >= 170
puts 175 >= 170
puts 170 <= 170
puts 170 <= 175
```

Entrada e saída

Aprendemos a ler do teclado (entrada, *input*) invocando o `gets` , e a imprimir um resultado na saída com o `puts` . Entrada e saída é vital para um programa pois é a maneira padrão dele se comunicar com o mundo exterior. Por enquanto a leitura é

feita do teclado (na verdade, da entrada padrão), e a saída para o console (na verdade, a saída padrão). No decorrer deste curso veremos como "ler de" e "escrever para" um arquivo. No futuro será possível, por exemplo, "ler da" ou "enviar informações para" a internet, ler dados de um joystick ou se comunicar com uma caixa de som via bluetooth. Tudo isso é entrada e saída: o joystick é uma entrada de dados, uma caixa de som é uma saída.

Invocando funções

Ao lermos e escrevermos dados, utilizamos duas palavras importantes: `gets` e `puts`. Essas duas palavras não são meras palavras, são algo que podemos chamar, invocar, pedir para ser executado. São **funções**. Uma função pode receber parâmetros, como no caso de `puts`:

```
puts "Bem vindo ao jogo da adivinhação"
```

Em Ruby, ao invocar uma função, o uso do parênteses é opcional em muitas situações, o caso a seguir mostra o mesmo código agora com parênteses:

```
puts("Bem vindo ao jogo da adivinhação")
```

Desenvolvedores Ruby costumam invocar funções sem o uso do parênteses, que é o padrão que seguiremos sempre que ficar claro o que está acontecendo.

Parâmetros e retorno de uma função

Vimos também que em Ruby é possível ter uma função com parâmetro opcional, que é o próprio caso de `puts`. Os exemplos a seguir tem todos o mesmo resultado: a impressão de uma linha em branco:

```
puts ""
puts
puts("")
puts()
```

Por fim, uma função retorna alguma coisa. Lembra da função de soma na matemática? O que ela retornava? A soma de dois números. E a função de multiplicação? Um número vezes o outro. O que a função de potência retornava? Um número multiplicado diversas vezes por ele mesmo. Toda função retorna algo em Ruby e, no caso de `gets` ele retorna uma linha de entrada do usuário. Podemos fazer o que quiser com esse retorno, como atribuí-lo a uma variável:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "Começaremos o jogo para você, " + nome
```

Mas o retorno de uma função já pode ser usado direto para invocar outra função:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
puts "Começaremos o jogo para você, " + gets
```

Código estranho né? Temos que tomar cuidado com código estranho. Se está estranho para nós, para quem ler este código daqui dois meses estará mais estranho ainda. Por isso mesmo que não o escrevemos. Nosso objetivo como programadores não é escrever um código indecifrável, é escrever um código que funcione e que possa ser alterado no futuro sem criar bugs. E é isso que buscaremos no decorrer deste curso, aprender juntos a programar e escrever bons programas.

O que é refatoração: exemplo da quebra de linha

Ao implementar nosso jogo escrevemos seis `puts` só para pular seis linhas. É hora de parar, olhar para trás e melhorar nosso código. Este é o processo que chamamos de *refatoração*. Queremos alterar nossas seis linhas de `puts` para um código mais simples que alcance o mesmo resultado. Seria muito bom se fosse possível escrever:

```
puts "proxima_linha,proxima_linha,proxima_linha,proxima_linha,proxima_linha,proxima_linha"
```

O Ruby não entende isso. Mas existe uma sequência que significa uma `n` ova linha, *new line*, a sequência `\n`. Portanto, podemos substituir nosso código por:

```
puts "\n\n\n\n\n\n"
```

O resultado é o mesmo. Se isso funciona, podemos substituir também o `puts` de quatro linhas por quatro `\n`s. Nosso código agora está bem mais simples e compacto:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"
puts "\n\n\n\n"
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute
puts 175 == chute.to_i
```

Interpretador ou Compilador

Cada arquitetura de computador e sistema operacional possui um determinado conjunto de comandos, uma linguagem, que ele entende. Alguns dos sistemas compartilham os mesmos comandos, mas mesmo assim são comandos muito básicos, que permitem basicamente operações matemáticas (simples e complexas), e teríamos que implementar todo um programa

ou jogo com esses comandos se desejássemos utilizar sua linguagem. Não só isso, é necessário escrever para cada uma das arquiteturas de máquina e sistema operacional, um trabalho enorme. Escrever nessas linguagens mais baixo nível, mais próximas da linguagem da máquina (código de máquina) é, portanto, raro.

Na prática existe um mercado muito maior para desenvolvedores que escrevem em uma linguagem mais alto nível, com menos preocupações tão pequenas. Nesse mundo, temos dois caminhos famosos. Podemos escrever nosso programa em forma de texto, como viemos fazendo até agora, e rodar um outro programa que lê o nosso e traduz, interpreta, em temo real para que o computador execute os comandos que desejamos. Nesse sentido, temos que instalar primeiro esse interpretador, que existe uma versão para cada arquitetura e sistema operacional, mas somente uma versão do nosso código. Esse é o processo de interpretador, que a implementação padrão da linguagem Ruby utiliza.

Ou podemos pedir para um programa transformar o nosso texto, escrito uma única vez, na linguagem da máquina, o que geraria um programa auto suficiente, executável. Ele compila um programa para cada configuração de máquina desejada. Portanto essa outra abordagem, de um compilador, acaba gerando diversos arquivos executáveis. A linguagem C é famosa por ser compilada.

Existem também linguagens híbridas e diversas outras variações. A linguagem mais próxima e famosa de código de máquina, mas que não é código de máquina, é o Assembly Language.

A linguagem Java é um exemplo de variação, é uma linguagem primeiro compilada (arquivos *.class*), depois interpretada (*Java Virtual Machine*) e, por fim, compilada para código de máquina (*just in time developer*).

O JRuby, um interpretador do Ruby baseado em Java, também se baseia na mesma abordagem híbrida.

Continuando a escrever o jogo: `if` e `else`

Escrevemos o início de nosso jogo, onde o jogador é capaz de chutar uma única vez um número entre 0 e 200 e dizemos se ele acertou ou não. Um jogo praticamente impossível de se ganhar. Mais ainda se a única informação que damos é `true` ou `false`. Primeiramente, vamos mudar essa mensagem para dizer algo como `Acertou!` ou `Errou!`. Isto é, trocamos a linha do `puts 175 == chute.to_i` por duas linhas que imprimem essas mensagens:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets

puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome

puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"

puts "\n\n\n\n"
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

puts "Acertou!"
puts "Errou!"
```

Mas agora o programa não faz mais sentido nenhum: ele sempre imprime as duas mensagens! Queremos imprimir a primeira mensagem somente caso o nosso `numero_secreto` seja igual ao valor chutado (`chute.to_i`). E só vamos imprimir a mensagem de erro do chute caso contrário. Isto é, gostaríamos de fazer algo como:

```
se numero_secreto == chute.to_i
    puts "Acertou!"
se não
    puts "Errou!"
fim
```

Infelizmente o Ruby não usa "se" e "se não" como termos-chave para fazer o código acima funcionar. Mas ele tem algo muito parecido: o `if` e o `else` . Eles dois, junto com o `end` , permitem definir dois blocos de código que serão executados somente a condição verificada seja verdadeira (`if`) ou falsa (`else`), portanto se alterarmos nosso programa para:

```
if numero_secreto == chute.to_i
    puts "Acertou!"
else
    puts "Errou!"
end
```

Rodamos nosso jogo e temos o resultado adequado quando erramos:

```
...
Tentativa 1
Entre com o numero
160
Será que acertou? Você chutou 160
Errou!
```

If sem else: será?

É possível usar o `if` sem o `else` também. A segunda condição é opcional, como em:

```
if numero_secreto == chute.to_i
    puts "Acertou!"
end
```

Sempre que temos um `if` sem `else` vale a pena se perguntar o que acontecerá no caso contrário. Queremos mesmo ignorar a outra condição? Em nosso exemplo, não queremos, e em muitos exemplos do dia-a-dia queremos mostrar alguma mensagem de erro. Mas existem situações onde é natural usarmos somente a condição `if` . É importante sempre pensar e pesar qual abordagem deseja utilizar. O exemplo a seguir mostra uma situação em que não utilizaríamos o `else` sem problema nenhum:

```
if numero_foi_muito_proximo
    put "Quase lá! Está chegando bem perto!"
end
```

Code smell: comentários

Só que nosso código está meio feio, essa linha do `if` está fazendo bastante coisa, comparando um número com o valor de uma *string* transformada em inteiro. Quanta coisa! Para deixar claro o que está acontecendo vamos comentar uma linha de nosso código: um comentário é um texto qualquer que será ignorado pelo interpretador do Ruby e usamos a `#` para fazê-lo:

```
# verificando se acertou
if numero_secreto == chute.to_i
    puts "Acertou!"
else
    puts "Errou!"
end
```

Mas cuidado, comentários são indícios que nosso código está difícil de entender, o código está "fedido", "*the code smells*", em inglês. Nesse instante nosso código até é razoável, mas se existe a necessidade de comentá-lo porque não entendemos direito, é hora de melhorar nosso código. É hora de refatorá-lo. Já vimos um tipo de refatoração antes, agora veremos um novo.

Temos uma condição `numero_secreto == chute.to_i`. O que essa condição nos diz? Se acertei ou não. Portanto que tal extrairmos dai o seu valor em uma nova variável (*extract variable*)?

```
acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    puts "Errou!"
end
```

Olha como nosso `if` ficou muito mais claro. No fim, temos o texto que diz exatamente quando o código será executado: "if acertou", isto é, se acertou, imprime `Acertou!`.

Outros tipos de comentários

Existem diversos tipos de comentários em Ruby. O mais famoso é o de uma linha, como já vimos:

```
# meu nome
nome = "Guilherme"
```

Ou o comentário de múltiplas linhas:

```
=begin
 comentário de
 varias linhas
=end
```

Mas o comentário de múltiplas linhas mais comum é o uso do próprio `#` em todas elas:

```
#  comentário de
#  varias linhas
nome = "Guilherme"
idade = 33
```

Existem ainda outras maneiras de se comentar código em um arquivo, mas o mais importante é lembrar que se precisa comentar, pode ser que precise refatorar. Repense seu código.

Condições aninhadas (*nested ifs*)

Mas o jogo ainda é muito difícil. Queremos deixá-lo mais fácil dizendo para o jogador, caso ele erre, se o número é maior ou menor do que o número chutado. Já conhecemos o operador de menor (`<`) e de maior (`>`), e as condições que podemos criar com `if`s, portanto podemos verificar se o número é maior e dar a mensagem de acordo:

```
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
```

Mas só queremos executar esse código caso o jogador não tenha acertado, então:

```
acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end
```

O jogo funciona e a estrutura que criamos, com um `if` dentro de outro `if`, é chamada de *nested if*. Repare que esse tipo de estrutura é um sinal de lógica mais complexa, afinal o código está fazendo diversas verificações, não só uma como fazia antigamente.

A variável diferença

Que tal criar uma variável chamada diferença? Algo como:

```
diferenca = numero_secreto - chute.to_i
```

Poderíamos agora verificar se a diferença é zero (acertou), maior (o número secreto é maior) ou menor (o número secreto é menor) que zero. É uma abordagem válida mas ainda manteria duas ou três condições em nossos `if`s, portanto não utilizaremos essa abordagem nesse instante.

Code smell: *copy* e *paste*

Mesmo assim o jogo continua difícil. Falta um último passo para permitir que nosso jogador tenha alguma chance de ganhar: temos que dar algumas chances, algumas "vidas", para que ele possa tentar. Vamos dar 3 vidas, 3 tentativas, portanto fazemos um *copy* e *paste* do código da tentativa, mudando apenas o número dela:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"

puts "\n\n\n\n"
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end

puts "\n\n\n\n"
puts "Tentativa 2"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
```

```
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end

puts "\n\n\n\n"
puts "Tentativa 3"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end
```

Agora sim, jogamos e acertamos na terceira tentativa:

```
...
Tentativa 1
Entre com o numero
100
Será que acertou? Você chutou 100
O número secreto é maior!
...
Tentativa 2
Entre com o numero
180
Será que acertou? Você chutou 180
O número secreto é menor!
...
Tentativa 3
Entre com o numero
175
Será que acertou? Você chutou 175
Acertou!
```

O jogo funciona e já tem uma certa graça, pois é desafiador acertar em três tentativas. Mas agora quero corrigir a mensagem de entrada do número, repare que esqueci de colocar acento em "numero". Tenho que mudar o código em três lugares diferentes pois eu fiz *copy* e *paste*. Isto não é bom, não foi uma boa prática. Veremos como melhorar esse código para dar andamento em nosso jogo!

O laço `for` (*loop*)

Nosso jogo atual pode divertir por alguns minutos mas é muito fácil perceber que será difícil de manter seu código a medida que evoluirmos o mesmo. *Copy* e *paste* é uma das práticas mais nocivas a um código e abusamos dele aqui de propósito. Queremos executar o mesmo pedaço de código três vezes sem precisar usarmos essa má prática. É hora de refatorar.

Seria muito bom poder dizer para o interpretador algo como:

```
execute 3 vezes
    puts "\n\n\n\n"
    puts "Tentativa 1"
    puts "Entre com o numero"
    chute = gets
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Note que já adicionei um `end` para dizer para o interpretador onde nosso código a ser executado diversas vezes termina.

Mas a linguagem não sabe o que signifca "3 vezes". O que ela conhece são variáveis, números, etc. Como representar então 3 vezes em números? Da mesma maneira que uma criança representa os três números: 1, 2 e 3! Imagine Julieta, dois aninhos, para quem pedimos, por favor conte até três. E ela conta: `1`, `2` e `3`. O que ela fez? Ela criou uma variável chamada dedos, e colocou o número um lá dentro. Depois o número dois na variável dedos, e por último o número três. Podemos mandar o computador fazer a mesma coisa: crie uma variável chamada `dedos` e execute o código a seguir para todos os números inteiros entre 1 e 3:

```
rode para dedos em 1 ate 3
    # codigo a ser executado
end
```

Estamos quase lá. Novamente, Ruby não fala português tão bem quanto a Julieta, então traduzimos para a linguagem dele: `1 até 3` vira `1..3`, `rode para` vira `for` e `em` vira `in`:

```
for dedos in 1..3
    # codigo a ser executado
end
```

Podemos testar com uma mensagem de impressão simples:

```
puts "Vou contar de 1 até 3"

for dedos in 1..3
    puts "Contando"
end
```

Resultando em:

```
Vou contar de 1 até 3
Contando
Contando
Contando
```

Só faltou contar de verdade, isto é, faltou imprimirmos o número que estamos contando. Dentro do código a ser executado diversas vezes temos acesso a variável chamada `dedos`, que é o valor que estamos passando agora:

```
puts "Vou contar de 1 até 3"

for dedos in 1..3
    puts "Contando: " + dedos
end
```

Mas ao rodar, percebemos que temos um erro de execução, algo ligado a não ser possível converter um número em uma `String`:

```
TypeError: no implicit conversion of Fixnum into String
```

Claro, não faz sentido somar um texto com um número. Faz sentido concatenar duas *Strings*, ou somar dois números. Mas um número e uma *String* não podem ser concatenados nem somados em Ruby. Portanto, vamos converter nosso número inteiro em uma *String* invocando seu método que transforma em *String*, o `.to_s`:

```
puts "Vou contar de 1 até 3"

for dedos in 1..3
    puts "Contando: " + dedos.to_s
end
```

Resultando em:

```
Vou contar de 1 até 3
Contando: 1
Contando: 2
Contando: 3
```

Aplicando o laço `for` no nosso programa

Pronto! O que temos aqui é um trecho de código que será repetido diversas vezes. Um *loop*, um laço que executa nosso código três vezes. O `for` é um dos diversos tipos de *loop* que podemos fazer e veremos outras maneiras mais pra frente.

Queremos nos livrar da sujeira, executando o mesmo trecho três vezes. Para isso criamos um laço cuja variável `tentativa` vale entre `1` e `3`:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"


for tentativa in 1..3
    puts "\n\n\n\n"
    puts "Tentativa 1"
    puts "Entre com o numero"
    chute = gets
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Agora nosso programa roda três vezes. Mas sempre imprime `Tentativa 1`. Corrigimos para mostrar a tentativa certa junto com o total de tentativas:

```
# ...

for tentativa in 1..3
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de 3"
    puts "Entre com o numero"
    chute = gets
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i
```

```
    if acertou
        puts "Acertou!"
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Nosso jogo continua funcionando como antes, mas já sabemos que é bem mais fácil mantê-lo agora. Podemos corrigir a mensagem do `numero` sem acento trocando um único ponto do programa, pois não temos mais o *copy* e *paste*:

Mas 3 tentativas parece ser muito pouco para acertar. Queremos mudar para 5. Alteramos no nosso laço:

```
for tentativa in 1..5
```

E rodamos. Por mais que o jogo me dê cinco tentativas, ele imprime:

```
...
Tentativa 1 de 3
...
Tentativa 2 de 3
...
Tentativa 3 de 3
...
Tentativa 4 de 3
...
Tentativa 5 de 3
...
```

Onde errei?

Code smell: Magic numbers

Repare que o número "3" apareceu duas vezes em nosso código, tanto no laço quanto na impressão da mensagem de qual era a tentativa atual. O que há de errado nisso? Primeiro que nosso número está duplicado. Antes vimos um exemplo de *copy* e *paste* de um grande trecho de código, e percebemos que mudar um ponto seria trabalhoso pois teríamos que mudar todos.

Mas repare que mesmo quando temos um simples número repetido em duas partes do meu programa é muito fácil esquecer que ele existe. Também seria inviável procurar todas as partes de todos os meus arquivos onde o número *3* aparece. Ele quer dizer muitas coisas ,e justamente por isso, ele é um número mágico jogado no meu programa: ninguém sabe o que ele é, e ele aparece jogado em diversos lugares.

O que faremos então é dar um nome a esse número, criamos então uma variável que se chama `limite_de_tentativas` :

```
limite_de_tentativas = 3
```

Efetuamos a refatoração e extraímos nossa variável. Agora nosso número não é mais mágico, ele é muito bem definido e está claro para todos o que significa. Após extrair a variável, refatoramos quem acessava esse valor (no caso, mágico) e substituímos por uma referência a variável que descreve seu valor:

```
for tentativa in 1..limite_de_tentativas
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " + limite_de_tentativas.to_s
```

Pronto! Agora basta mudar o limite para `5`:

```
# ...

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    # ...
end
```

Quebrando o laço com o *break*

Ótimo! Para terminarmos nossa primeira versão do jogo temos que corrigir um único bug, nosso primeiro bug detectado pelo jogador. Se ele acertar de primeira o número `175`, o jogo continuará perguntando, feito bobo. Queremos alterar nosso jogo para, caso o usuário acerte, ele saia fora do laço, ele "quebre" (*break*) o nosso laço (*loop*). Algo como:

```
    if acertou
        puts "Acertou!"
        quebra
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
```

E adivinha, o Ruby possui uma palavra chave chamada `break` que quebra o laço atual, saindo dele completamente! Fazendo a alteração a seguir, o jogo termina mesmo que o jogador acerte de primeira:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
```

```
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " + limite_de_tentativas.to_s
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
        break
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Com o resultado:

```
...
Tentativa 1 de 5
Entre com o número
175
Será que acertou? Você chutou 175
Acertou!
```

Resumindo

Vimos nesta aula:

como funciona o operador `+`, que é válido entre *Strings*, quando executa uma operação de concatenação, e entre números, ao executar uma soma como converter um número inteiro em `String`, como executar código condicionalmente com o uso do `if...else...end`, a utilização e o pepino de se usar `if`s aninhados, como funciona e (novamente) o mal cheiro que um comentário indica, a refatoração de extrair variável, o controle de fluxo através de um laço, permitindo a execução repetida de um trecho de código, além de como quebrar esse laço a qualquer instante com o uso do `break`. Nosso jogo já nos desafia a acertar um número entre 0 e 200 de maneira desafiadora (pode testar, é difícil alguém acertar em tão poucas tentativas). Nosso próximo passo é melhorar bastante esse nosso código antes de adicionar diversas novas funcionalidades.

Avançar