10

Lendo, escrevendo em arquivos e entendendo a pilha de execução

Entrada e saída de arquivo: palavras aleatórias e o top player

Lendo um arquivo de palavras, nosso dicionário

Chegamos aos nossos últimos desafios do jogo da forca. Primeiro queremos fazer com que a lista de palavras seja lida de um arquivo, para que o jogador não tenha idéia de qual palavra estamos utilizando.

Para isso utilizaremos um arquivo bem simples, chamado `dicionario.txt`, cujo formato é uma palavra por linha:

```
alura
casa do codigo
caelum
desenvolvedor
programador
software
refatorar
code smell
```

Como ler então os dados de um arquivo? Existem diversas maneiras de executar essa tarefa. Uma delas lê todo o conteúdo do arquivo de uma única vez para a memória:

```
texto = File.read("dicionario.txt")
```

Podemos depois quebrar o texto em uma array de `String`s separadas pelas quebras de linha, para isso separamos (::split::) pelas quebras de nova linha (::\n::):

```
texto = File.read("dicionario.txt")
todas_as_palavras = texto.split("\n")
```

Agora que temos um array de palavras podemos escolher um número entre `0` e o total de palavras:

```
texto = File.read("dicionario.txt")
todas_as_palavras = texto.split("\n")
numero_aleatorio = rand(todas_as_palavras.size)
```

E escolher a palavra secreta:

```
texto = File.read("dicionario.txt")
todas_as_palavras = texto.split("\n")
```

```
numero_aleatorio = rand(todas_as_palavras.size)
palavra_secreta = todas_as_palavras[numero_aleatorio]
```

Pronto, basta colocarmos esse código em nossa função `sorteia_palavra_secreta`. Mas essa função está em `UI`, não em lógica:

```
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 então duas funções de `UI`, `avisa_escolhendo_palavra` e `avisa_palavra_escolhida`:

```
def avisa_escolhendo_palavra
    puts "Escolhendo uma palavra..."
end

def avisa_palavra_escolhida(palavra_secreta)
    puts "Escolhida uma palavra com #{palavra_secreta.size} letras... boa sorte!"
    palavra_secreta
end
```

Separamos então uma função de lógica `sorteia_palavra_secreta`:

```
def sorteia_palavra_secreta
    avisa_escolhendo_palavra
    texto = File.read("dicionario.txt")
    todas_as_palavras = texto.split("\n")
    numero_aleatorio = rand(todas_as_palavras.size)
    palavra_secreta = todas_as_palavras[numero_aleatorio]
    avisa_palavra_escolhida palavra_secreta
end
```

Limpando a entrada de dados

Os dados que estamos lendo de nosso arquivo ainda podem possuir algumas sujeiras. Por exemplo, não queremos reclamar que o jogador errou caso ele chute a letra `P` maiúscula para a palavra `programador`, ou ainda que a palavra esteja registrada como `progrAmAdor` e que isso faça com que o chute `a` erre.

Precisamos limpar, padronizar, normalizar, nossa entrada de dados. Podemos começar com nosso leitor de chutes, onde simplesmente transformamos o chute para letras minúsculas invocando o método `downcase`:

```
def pede_um_chute
    puts "Entre com a letra ou palavra"
```

```
    chute = gets.strip.downcase
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Ao ler nosso arquivo também devemos utilizar minúsculos para garantir que o chute, agora sempre em minúsculo, encontre sua posição correta nas palavras secretas. Para isso podemos ao escolher a palavra secreta transformá-la em minúsculo:

```
def sorteia_palavra_secreta
    avisa_escolhendo_palavra
    texto = File.read("dicionario.txt")
    todas_as_palavras = texto.split("\n")
    numero_aleatorio = rand(todas_as_palavras.size)
    palavra_secreta = todas_as_palavras[numero_aleatorio].downcase
    avisa_palavra_escolhida palavra_secreta
end
```

Pronto! Adicione no seu dicionário a palavra `RuBy` e teste a mesma com um `r` e `Y`, ambos devem funcionar. Você pode "roubar" e deixar temporariamente somente esta palavra no dicionário para forçar o teste a sair a palavra que queria.

Processamento e memória devem ser otimizadas?

Por mais que essa abordagem funcione e seja totalmente válida, ler um arquivo inteiro para a memória pode ser ruim. Temporariamente consumimos muita memória e também toda vez que invocamos a função para sortear uma nova palavra temos que passar por todo o arquivo, mesmo que a palavra sorteada seja, por exemplo, a primeira. Imagine que se o arquivo possui diversos megas, ocuparemos todo esse espaço na memória para sortear uma única palavra. Uma outra opção seria ler primeiro um número no arquivo, indicando quantas palavras existem:

```
8
alura
casa do codigo
caelum
desenvolvedor
programador
software
refatorar
code smell
```

Após ler o número de linhas existentes:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
```

Agora podemos sortear um número:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
aleatoria = rand(total_de_palavras)
```

Agora vamos até a linha adequada, ignorando diversas delas:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
aleatoria = rand(total_de_palavras)
for i in 1..aleatoria -1
    arquivo.gets
end
```

Finalmente lemos a palavra secreta, limpando ela, e fechamos o arquivo:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
aleatoria = rand(total_de_palavras)
for i in 1..aleatoria -1
    arquivo.gets
end
palavra_secreta = arquivo.gets.strip.downcase
arquivo.close
```

Cada abordagem de leitura de arquivo implica em um consumo de memória e processamento maior ou menor. Nesse curso nosso foco não é otimização, e um dicionário de 1 mega de palavras não faz cócegas aos computadores modernos, não precisamos nos preocupar com isso. Claro, no seu devido tempo, a medida que entramos em assuntos como otimização e análise de algoritmos essa preocupação aumenta, mas sempre questionando se ela faz sentido ou não. Aqui não há diferença prática no resultado.

Existem ainda outras formas de acesso a arquivo em disco, como o acesso a qualquer posição do mesmo, sem precisar necessariamente ler as primeiras linhas (os primeiros bytes), mas para fazer isso você deve saber exatamente para que ponto do arquivo quer pular (em quantidade de bytes). Cada forma de acesso a um arquivo possui uma vantagem e desvantagem, como quase toda funcionalidade.

Escrita para arquivo: o melhor jogador

Devemos guardar quem foi o melhor jogador até agora, como forma de desafio ao próximo jogador. Primeiro vamos acumular os pontos por rodada. A função `joga` pode retornar os pontos até agora:

```
def joga(nome)
    # ...
    avisa_pontos pontos_ate_agora
    pontos_ate_agora
end
```

E ao invocarmos em nosso `jogo_da_forca` devemos acumular esses pontos, começando com zero:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    loop do
        pontos_totais += joga nome
        break if nao_quer_jogar?
    end
end
```

Precisamos avisar o jogador toda vez quais são seus pontos totais, portanto definimos a função em `ui.rb` :

```
def avisa_pontos_totais(pontos)
    puts "Você possui #{pontos} pontos."
end
```

E a cada nova rodada mostramos esses pontos para o jogador:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais
        break if nao_quer_jogar?
    end
end
```

Pronto, ao jogarmos nosso jogo e acertando as duas primeiras vezes temos o resultado de `200` pontos acumulados:

```
...
Será que acertou? Você chutou refatorar
Parabéns! Acertou!
Você ganhou 100 pontos.
Você possui 200 pontos.
Deseja jogar novamente? (S/N)
```

Antes de perguntarmos se o usuário deseja jogar novamente devemos salvar o nome do usuário e seus pontos em um arquivo, algo como:

```
def jogo_da_forca
    nome = da_boas_vindas
```

```
    pontos_totais = 0

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais
        salva_rank nome, pontos_totais
        break if nao_quer_jogar?
    end
end
```

E definimos a função `salva_rank`:

```
def salva_rank(nome, pontos)
end
```

Definimos o conteúdo do arquivo, que será o nome na primeira linha e os pontos na segunda:

```
def salva_rank(nome, pontos)
    conteudo = "#{nome}\n#{pontos}"
end
```

Por fim, utilizamos a função `write` de `File` para salvar o conteúdo em um arquivo chamado `rank.txt`:

```
def salva_rank(nome, pontos)
    conteudo = "#{nome}\n#{pontos}"
    File.write "rank.txt", conteudo
end
```

Rodamos nosso jogo e após vencer duas vezes com `200` pontos temos a saída no arquivo:

```
guilherme
200
```

Lendo o melhor jogador

Queremos mostrar para o jogador atual quem é nosso campeão, e quantos pontos ele já conquistou. Já sabemos ler os dados de um arquivo e quebrar as linhas:

```
def le_rank
    conteudo_atual = File.read "rank.txt"
    dados = conteudo_atual.split("\n")
end
```

Agora que temos os dados do rank podemos mostrá-lo no começo do programa:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    avisa_campeao_atual le_rank

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais
        salva_rank nome, pontos_totais
        break if nao_quer_jogar?
    end
end
```

E definimos nossa função de `ui` que avisa o campeão atual:

```
def avisa_campeao_atual(dados)
    puts "Nosso campeão atual é #{dados[0]} com #{dados[1]} pontos."
end
```

Mas sempre salvamos o nome do último jogador e não é isso que queremos fazer. Desejamos sempre armazenar o melhor de todos os jogadores. Isto é, se o jogador já existe no arquivo, queremos manter somente o que possui mais pontos.

Antes de salvar temos que verificar se o nome e a pontuação que já estão lá são mais baixas que a pontuação atual. Portanto antes de salvar os dados devemos verificar sua pontuação:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    avisa_campeao_atual le_rank

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais

        if le_rank[1].to_i < pontos_totais
            salva_rank nome, pontos_totais
        end

        break if nao_quer_jogar?
    end
end
```

Pronto! Já temos nosso rank de primeiro lugar. Sempre armazenamos quem é o melhor jogador.

Refatoração: extrair arquivo

Só falta organizar um pouco mais nosso código. Repare que temos duas funções de lógica ligadas ao rank. Na verdade mais do que lógica, elas são as funções de armazenamento de dados (::data access::), portanto isolaremos tanto a `le_rank` quanto a `salva_rank` em um arquivo chamado `rank.rb`.

```
def le_rank
    conteudo_atual = File.read("rank.txt")
    dados = conteudo_atual.split("\n")
end

def salva_rank(nome, pontos)
    conteudo = "#{nome}\n#{pontos}"
    File.write("rank.txt", conteudo)
end
```

Não podemos esquecer de alterar nosso arquivo `forca.rb` para incluir relativo nosso `rank`:

```
require_relative 'ui'
require_relative 'rank'

# ...
```

Extrair código para outro arquivo não é uma refatoração grandiosa. Ela simplesmente varre para outro lugar o nosso código (que continua com funções globais), organizando em pequenas unidades (arquivos) para facilitar encontrar e manter as funções. Veremos em um próximo jogo como organizar ainda mais esse comportamento, fugindo da abordagem global.

A pilha de execução

Remova seu arquivo `rank.txt`. Agora rode novamente o jogo. Logo após entrar com seu nome, a aplicação para:

```
Começaremos o jogo para você, Guilherme
rank.rb:2:in `read': No such file or directory - rank.txt (Errno::ENOENT)
    from rank.rb:2:in `le_rank'
    from forca.rb:82:in `jogo_da_forca'
    from main.rb:3:in `<main>'
```

Como toda mensagem de erro, olhemos com calma o que acontece aqui. Primeiro, ele indica que o problema aconteceu no arquivo `rank.rb`, linha 2 (`rank.rb:2`). Ao tentarmos invocar a função `read (in read')`, `ocorreu um erro com a descrição` No such file or directory - rank.txt (Errno::ENOENT)`.

É fundamental entendermos a mensagem de erro para descobrir o que aconteceu. Ela diz que não foi encontrado o arquivo ou diretório `rank.txt`. Faz sentido, ele realmente não existe. Note que uma mensagem de erro realmente é rica de informações, ela nos em que arquivo, qual linha e qual a descrição que deu problema.

Mas o que são as informações que vem logo depois dela? Temos mais três linhas para entender melhor:

```
from rank.rb:2:in `le_rank'
from forca.rb:82:in `jogo_da_forca'
from main.rb:3:in `<main>'
```

A próxima linha indica em que arquivo e linha estávamos quando o erro ocorreu:

```
conteudo_atual = File.read("rank.txt")
```

A próxima linha que ele indica é no arquivo de lógica:

```
avisa_campeao_atual le_rank
```

Por fim, a linha do arquivo `main.rb` :

```
jogo_da_forca
```

O que essas linhas indicam? Repare que a linha 3 do `main.rb` é quem chama a função `jogo_da_forca` , que é quem chama a função `le_rank` , que é quem chama `File.read` que é onde acontece o erro. Isto é, essas linhas da mensagem de erro indicam o caminho que a execução do programa percorria quando do momento do erro. A única coisa "estranha" é que ela está de ponta-cabeça:

```
from rank.rb:2:in `le_rank'
from forca.rb:82:in `jogo_da_forca'
from main.rb:3:in `<main>'
```

Já simulamos nosso programa uma vez, vamos simulá-lo novamente. Agora com ainda mais carinho e atenção. Primeiro rodamos o comando `ruby main.rb` , que carrega o código do arquivo `main.rb` :

```
require_relative 'logic'

jogo_da_forca
```

A primeira linha carrega o arquivo de lógica, que por sua vez carrega os arquivos de `rank` e `ui` , todos definindo uma dezena de funções, mas nenhum executando algum código.

Então o programa chega até a terceira linha de nosso `main.rb` :

```
jogo_da_forca
```

Mas como o programa sabe em que linha ele está? Assim como uma criança precisa de seus dedos, uma variável, para saber em que número está, o programa também precisa de variáveis para indicar onde está no nosso código. Portanto, em algum lugar da memória, o programa armazena onde ele está:

```
funcao_atual = "main" # funcao sem nome, o código em si
arquivo_atual = "main.rb"
linha_atual = 3
```

Isto é, agora ele vai executar a linha 3. Mas ao tentar executar essa linha ele se depara com a invocação de uma função, ele precisa então ir até onde está essa função, no arquivo `forca.rb`, linha `78`:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    avisa_campeao_atual le_rank

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais

        if le_rank[1].to_i < pontos_totais
            salva_rank nome, pontos_totais
        end

        break if nao_quer_jogar?
    end
end
```

O programa troca então o conteúdo das variáveis `arquivo_atual`, `linha_atual` e `funcao_atual`:

```
funcao_atual = "jogo_da_forca"
arquivo_atual = "forca.rb"
linha_atual = 78
```

A linha 78 é a definição da função, que não faz nada. Mas ele logo executa a linha 79, 80 e 81, chegando na 82 com a seguinte memória:

```
nome = "guilherme"
pontos_totais = 0

funcao_atual = "jogo_da_forca"
arquivo_atual = "forca.rb"
linha_atual = 82
```

Agora ele invoca a função `le_rank`, trocando o valor das variáveis novamente:

```
nome = "guilherme"
pontos_totais = 0

funcao_atual = "le_rank"
arquivo_atual = "rank.rb"
linha_atual = 1
```

Imagine que o código da função `le_rank` funcione normalmente, que o arquivo exista. Nesse caso, ao sair da função `le_rank`, para onde o programa deve ir? Para o resto da linha `82` no `forca.rb` que é de onde o programa veio e invocou `le_rank`. Mas essa informação já era, já foi perdida quando trocamos o valor das variáveis para descrever `le_rank`. Opa. O programa ficaria perdido! Isso significa que algo que descrevi não como realmente um programa funciona. Ele não pode armazenar somente a linha e arquivo atual. Se ele trocar o valor, fim. O programa não sabe para onde voltar e continuar quando uma função retorna.

Vamos escrever o problema com todas as palavras: o programa precisa saber exatamente quem chamou quem, e quem chamou quem, e quem chamou quem, ..., e quem chamou quem, para saber para quem ele deve voltar, um a um. Isto é, podemos pensar que cada vez que chamamos uma função, avançamos um nível, cada vez que saímos de uma função, saimos desse nível. Por exemplo, considere o código a seguir no arquivo `nomes.rb`:

```
def le_nome
    nome = gets # 1
    puts "Lido!" # 2
    nome
end
def pede_nome
    puts "Digite seu nome" # 3
    nome_lido = le_nome # 4
    puts "Pedido!" # 5
    nome_lido
end
def inicio
    nome = pede_nome # 6
    puts "Bem vindo #{nome}" # 7

    puts "Quero conhecer mais alguém"
    nome2 = pede_nome # 8
    puts "Olá #{nome2}" # 9
end

inicio # 10
```

O interpretador do Ruby lê a definição das funções `le_nome`, `pede_nome` e `inicio`. Por fim, ele chega na linha marcada `10`, onde antes de invocar a função `inicio`, ele continua no código `principal` (::main::) de nosso arquivo. Portanto temos só um nível de chamada:

```
nomes.rb:10 in main
```

Ao invocar a função `inicio` ele vai para a linha marcada `6`, e anota o novo nível ::em cima:: do nível atual:

```
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Isto é, a informação acima nos diz exatamente como o programa chegou onde está. Continuemos então com a execução do código, invocando a função `pede_nome` e indo para a linha marcada `3` :

```
nomes.rb:3 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Ele agora imprime a mensagem `Digite seu nome` e chega na linha marcada 4.

```
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

```
    nome_lido = le_nome # 4
```

Essa linha possui dois passos. O primeiro é a invocação da função `le_nome` , e o segundo a atribuição do resultado dela a uma variável local chamada `nome_lido` . Agora ele invoca a função `le_nome` :

```
nomes.rb:1 in le_nome
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Note que agora leremos o nome do usuário, e chegamos na linha 2, portanto temos a existência de uma variável local:

```
def le_nome
    nome = gets # 1
    puts "Lido!" # 2
end
```

```
nomes.rb:2 in le_nome (nome="Guilherme")
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Agora ele sai da função, como fazer isso? Simples, basta jogar fora a última função que foi invocada, a última linha que foi ::empilhada:: (::push::) nessa pilha de funções (::stack::):

```
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Ainda existe o segundo passo a ser executado nessa linha. A função já retornou `"Guilherme"`, portanto é criada uma variável local chamada `nome_lido` com esse valor:

```
nomes.rb:4 in pede_nome (nome_lido = "Guilherme")
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Chegamos então na linha marcada `5`:

```
nomes.rb:5 in pede_nome (nome_lido = "Guilherme")
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Note que durante a execução dela, a variável antiga chamada `nome` já não existe mais, afinal, já saimos da função `le_nome`, mas a variável `nome_lido` local existe.

Terminando a execução da função `pede_nome`, o valor da variável `nome_lido` é retornado. Como fazemos mesmo a saída de uma função? Desempilhamos (::pop::) a última linha empilhada em nossa pilha de execução (::stack trace::).

```
nomes.rb:6 in inicio
nomes.rb:10 in main
```

A segunda parte da linha marcada `6` é a criação de uma variável chamada `nome`, com o valor do retorno da `pede_nome`:

```
nomes.rb:7 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

Imprimimos agora o `nome` com a mensagem de boas vindas. Continuamos a execução até a linha marcada `8`:

```
def inicio
    nome = pede_nome # 6
    puts "Bem vindo #{nome}" # 7

    puts "Quero conhecer mais alguém"
    nome2 = pede_nome # 8
    puts "Olá #{nome2}" # 9
end
```

```
nomes.rb:8 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

Entramos agora na função `pede_nome`, que entra na função `le_nome`:

```
nomes.rb:1 in le_nome
nomes.rb:4 in pede_nome
nomes.rb:8 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

```
def le_nome
    nome = gets # 1
    puts "Lido!" # 2
end
```

O que acontecerá após executarmos a linha marcada `1`? Teremos criado uma nova variável local com valor, por exemplo, `Daniela`:

```
nomes.rb:2 in le_nome (nome = "Daniela")
nomes.rb:4 in pede_nome
nomes.rb:8 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

Calma aí. Duas variáveis com o mesmo nome? Isso faz sentido? Claro! Elas são locais a função! Cada uma delas só é visualizada dentro de seu próprio escopo, no caso local. Então a variável chamada `nome` que está sendo utilizada na função `inicio` é uma variável totalmente diferente da variável chamada `nome` que está sendo utilizada na função `le_nome`.

Este é o ::local:: do nome variável ::local::. A variável vive e existe somente durante aquela chamada da função, e tem seu espaço de memória isolado da de outro método que foi chamado.

Assim como uma empilhadeira, a execução de um programa empilha as diversas funções que vão sendo invocadas. O processo de empilhar (::push::) e desempilhar (::pop::) é feito toda vez que entramos e saímos de uma função, permitindo que o programa saiba exatamente onde está e de onde veio. A pilha de coisas também tem nome, é a pilha de execução (::execution stack::). E aquelas linhas que mostram onde estamos, que nos mostra todo o caminho percorrido (::trace::) na pilha de execução é o ::stack trace::.

O código ASCII

Adicionando o código ASCII no ui.rb

```
def da_boas_vindas
    puts "/****************/"
    puts "/ Jogo de Forca */"
    puts "/****************/"
```

```
    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


def desenha_forca(erros)
    cabeca = "   "
    corpo = " "
    pernas = "   "
    bracos = "   "
    if erros >= 1
        cabeca = "(_)"
    end
    if erros >= 2
        bracos = " | "
        corpo = "|"
    end
    if erros >= 3
        bracos = "\\|/"
    end
    if erros >= 4
        pernas = "/ \\"
    end

    puts "  _______       "
    puts " |/      |      "
    puts " |      #{cabeca}  "
    puts " |      #{bracos}  "
    puts " |       #{corpo}  "
    puts " |      #{pernas}  "
    puts " |              "
    puts "_|___           "
    puts

end
```

Reescreve o avisa_acertou_palavra

```
def avisa_acertou_palavra
        puts "\nParabéns, você ganhou!"
        puts

        puts "       ___________      "
        puts "      '._==_==_=_.'     "
        puts "      .-\\:      /-.    "
        puts "     | (|:.     |) |    "
        puts "      '-|:.     |-'     "
        puts "        \\::.    /      "
        puts "         '::. .'        "
        puts "           ) (          "
        puts "         _.' '._        "
        puts "        '-------'       "
```

```
        puts
end
```

O que falta agora é só chamar os métodos o desenha_forca no cabeçalho

```
def cabecalho_de_tentativas (chutes, erros, mascara)
    puts "\n\n\n\n"
    desenha_forca erros
    puts "Palavra secreta: #{mascara}"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
end
```

Resumindo

Aprendemos a fazer a leitura de um arquivo de textos, a limpar (normalizar) os dados lidos para evitar sujeira, e a escrever em um arquivo as informações do nosso melhor jogador. Refatoramos nosso código para extrair um arquivo com as responsabilidades de entrada e saída para disco e simulamos nosso programa para entender um conceito fundamental de toda linguagem de programação: a pilha de execução.