07

# Testes

### Transcrição

[00:00] Vamos fazer um teste? Eu vou remover o arquivo rank.txt e vou rodar o meu jogo. A primeira coisa que fazemos dentro do jogo é ler o rank, e o arquivo não existe. O que será que acontece? Ele logo de cara para a aplicação dizendo que o arquivo não existe para ser lido.

[00:48] Vamos dar uma olhada o que são as linhas que ele menciona. Primeiro, rank.rb linha 7. É quando chamamos a função read, a função que deu o erro. O outro é forca.rb linha 95. É onde fica o lê rank, que quando chegou na linha 7 chamou a função read, que é onde deu o earro. Também temos o main.rb linha 3. É o cara que chama o jogo da forca, que está no arquivo forca.rb, que chama na linha 95 o lê rank, que é quem chama o file.read.

[01:45] Isso está indicando para nós na ordem inversa quem chamou quem, até dar o erro. O que vamos fazer é simular nosso programa, como já fizemos anteriormente, e ver exatamente como isso acontece.

[02:15] Quando executamos nosso programa, chegamos na linha 1 do main.rb. Ainda não estamos em função nenhuma, estamos na função principal do programa, e é chamada de função_atual:main. O interpretador do Ruby pode manter isso na memória para saber onde ele está.

[03:05] Quando ele terminar de executar isso e ir para a linha 2, ele incluiu, leu e processou todas as definições de funções que estão no arquivo forca. Isso é, ele pegou esse arquivo e começou a ler linha a linha. Ele definiu várias funções para nós, mas executar mesmo ainda não executou nada.

[03:53] Nosso próximo passo é chamar a próxima linha, porque a linha 2 não faz nada. Chegando na linha 3, ele entra para poder executar tudo que está lá dentro. Ele está indo para a função atual, jogo da forca, que é a função definida no arquivo forca.rb, na linha 91. Só a definição não vai fazer nada muito mágico. Agora ele vai executar a primeira linha do forca.rb, a linha 92, que é a linha das boas-vindas, que está no ui.rb.

[05:21] Depois, na linha 33, ele vai imprimir mensagens, ler valores, preparar para retornar quando chegar na 38. Nela, ele vai retornar o nome que foi lido. Fomos executando nosso programa e entrando onde tínhamos que entrar. Nosso interpretador Ruby manteve na memória a função atual, a linha atual, o arquivo atual, assim conseguiu ir passando para a próxima linha.

[06:00] Quando ele chegou na linha 38, a função acaba na 39. Acabada a função de boas-vindas, para onde ele vai? O que vem depois? Não vem nada, porque você não armazenou nada. Aí você pensa que depois de dar boas-vindas ele vai voltar para o jogo da forca. Eu sei disso e você também, mas como o interpretador sabe se ele só armazenou na memória o arquivo, a linha e a função atual? Ele não pode só armazenar isso. Senão ele vai ficar perdido no primeiro momento que voltar de uma função. Quando ele volta de uma função, ele tem que voltar para a função anterior. Ele não pode só armazenar a função atual, a linha atual e o arquivo atual. Ele tem que armazenar também a função anterior. Todas as funções anteriores que foram chamadas têm que ser armazenadas para ele saber onde ele está, porque aí quando ele sai de uma função, ele volta para a função anterior na linha seguinte.

[07:15] Não adianta eu armazenar só a linha atual. Tenho que armazenar uma em cima da outra qual a próxima linha que foi executada, a próxima função. Uma em cima da outra. Quando uma função termina, eu tiro ela de cima e continuo executando a última. Preciso armazenar na memória todas as funções que estou invocando nesse instante até chegar no ponto atual, para depois poder continuar as anteriores.

[08:00] Essa simulação que fizemos, com essas variáveis, não é como o interpretador Ruby funciona, ou qualquer interpretador. Ele precisa de alguma maneira armazenar todas as funções que estão sendo invocadas até chegar no ponto em que estamos do nosso programa.

[08:16] Vamos criar então um programa mais simples para poder simular linha a linha entrando e saindo das funções. Vou criar um arquivo nomes.rb que vai pedir um nome. A primeira função que eu vou ter lê um nome. Ela vai fazer o nome = gets, puts sendo lido esse meu nome, devolvo o nome, e coloco o endereço. A segunda função é a que pede um nome. O nome lido vai ser igual a lê o nome e devolve o nome lido. Vamos ter uma última função, chamada de início. Ela vai falar que queremos um nome, dizer bem-vindo para esse nome, que ela quer conhecer mais alguém, e pedir o segundo nome. Por fim, mostrando uma mensagem de olá para o segundo nome.

[09:55] Meu código vai rodar essa função início e vemos tudo acontecendo com calma. Vamos simular esse código linha a linha e ver o que acontece, como o interpretador pode armazenar todas as invocações e saber para onde ele deve voltar quando uma função termina. Lembre que a função principal do nosso programa, quando carregamos e digitamos hub e o nome de um arquivo é chamada main. Quando chamarmos hubnomes.rb estaremos lendo o arquivo nomes.rb na linha 1, na função que chama main. Mas a linha um desse arquivo só define uma função. Ele não executa.

[10:50] Aí ele vai até a linha 6, onde ele define outra função. Na linha 12, onde define outra função. Chegando na linha 21 é onde ele vai executar a função início. É onde ele vai invocar a função. Eu vou em cima empilhar, tenho uma pilha de funções, colocar em cima um novo item, que é no arquivo nomes.rb estou indo para a linha 13 de execução, que é a linha da função início.

[11:37] Então, cheguei na 12, era só realmente não fazer nada. Cheguei na 13, invocando para valer a função início. Na linha 13, primeiro ele vai chamar a função pede nome. Depois ele vai atribuir o resultado dela à variável nome. Vamos empilhar uma invocação ao arquivo nomes.rb.

[12:07] Agora, o que ele faz dentro do pede nome? Primeiro, na linha 7 ele mostra a mensagem para digitar o nome. Aí, na linha 8, ele vai invocar uma função e atribuir o resultado à variável nome lido. Se ele vai invocar outra função, nós chamamos uma função. Empilhamos ela na linha 2 em lê nome.

[12:42] O tempo inteiro tenho exatamente como cheguei até a execução, nos traços da execução, o que fui deixando para trás à medida em que invocava, até chegar à execução. Estou empilhando todas as invocações da minha execução.

[13:00] A linha 2 invoca uma função get, devolve um resultado que é uma string, e depois atribui a variável nome. Quando ele chegar na linha 3, eu vou ter uma variável chamada nome, local, que vale Guilherme. Essa variável local só é visível dentro da nossa função lê nome. Agora, executo a linha 3, que é o puts lido com o nome. Ele mostra para mim lido Guilherme. E era o que eu queria.

[13:38] Agora, o que acontece? Ele vai para a linha 4. A linha 4 só vai devolver o nome, o valor da variável nome. Quando termino de invocar essa função, nós arrancamos esse último item da nossa pilha. Desempilhamos o último item da pilha de execução, retornando o valor Guilherme.

[14:13] Repare que a variável nome desapareceu. O valor dela existe, é o resultado da função lê nome, que agora na linha 8 é atribuído à variável nome lido. Então, nome lido é igual a Guilherme. E eu terminei de executar a linha 8. A variável nome não existe mais. Ela era local na nossa função lê nome. Agora, na função pede nome tenho outra variável que tem o mesmo valor, que é Guilherme. Valor é uma coisa que está na memória. O nome da variável é uma referência a esse valor. Posso ter várias variáveis evidenciando o mesmo valor. Nesse caso, o nome lido está referenciando a mesma string que o nome estava antes. Mas a variável nome não existe mais, ela era local, assim como nome lido também é local.

[15:14] Na linha 9 ele só imprime o pedido. Aí na linha 10 ele retorna o valor do nome lido. A 13 termina de executar. Ele imprime para nós e eu fico muito feliz de ter recebido a mensagem na linha 14. Ele vai para a linha 16, onde imprime que quer conhecer mais alguém. E então imprime essa mensagem, chegando na linha 17. Ainda tem a variável local chamada nome e vai invocar a função pede nome. Já conhecemos essa função. Ela vai começar na linha 7.

[16:14] Na linha 7 ele imprime digite o seu nome. Depois vai para a linha 2, que chama a função lê nome, e chama um gets. Eu digito o outro nome, e ele atribui à variável local o valor João.

[17:08] A variável nome local, a função lê nome, é uma coisa. A variável nome local, a função início, não tem nada a ver. Ela é local ali dentro. O valor até pode ser passado como parâmetro, mas o nome da variável só fica visível ali dentro. Não tem nada a ver uma variável com a outra. Essa variável local só vive durante essa execução dessa função. Ela é local à execução dessa função. Quando essa função acabar, já era o nome dessa variável. O valor dela pode até ser retornado.

[18:10] Repare que para podermos interpretar um programa para valer, não adianta armazenar só a última função que foi invocada, em que arquivo, em que linha está essa função. Temos que empilhar todas as invocações. À medida que elas vão terminando, vamos desempilhando essas invocações. À medida em que fazemos isso, estamos armazenando na memória uma pilha de execução. A pilha de execução do meu programa. E é essa pilha de execução que o Ruby mostrou para mim quando deu erro. A pilha de execução é uma maneira muito útil de tirar informação de como chegamos até determinado lugar, para poder tentar induzir o que aconteceu de errado no meio do caminho. É um conceito bem importante para executar os programas.

[19:07] Nós vemos ela acontecendo simulando, como fizemos agora, linha a linha, empilhando cada uma das funções, desempilhando à medida em que elas terminam. Não há segredo fora isso. Lembre que as variáveis locais sobrevivem só naquela execução, e é por isso que podemos ter diversos tipos de variáveis com nomes iguais, mas em escopos diferentes. Uma no escopo de uma variável local com uma variável local numa chamada de invocação de função, e uma outra com uma outra invocação de uma função. Não tem nada a ver uma com a outra. Cada uma só é visível na sua execução, naquele pedaço da pilha de execução.