



Trabalho de Programação Orientada à Objetos

Código de um jogo de quebra-cabeça tipo “Racha-Cuca”

Professor: Saulo Henrique Cabral Silva

Alunos: Lucas Cardoso Costa e Maria Luísa Matosinhos

Ouro Branco

2024

Sumário

Sumário.....	2
Introdução.....	4
Implementação.....	4
scan.....	4
encerraJogo().....	6
imprimeLinha().....	6
criarTabuleiro().....	7
imprimirTabuleiro().....	8
trocaPecas().....	10
embaralharTabuleiro().....	11
menu().....	15
lerOpcao().....	17
selecionarDificuldade().....	18
moverPeca().....	19
validarJogada().....	21
conferirTabuleiro().....	24
limparBuffer().....	25
encerrarPartida().....	26
comecarJogo().....	28
fecharPartida().....	31
continuarJogo().....	32
limparConsole().....	33
main().....	34
Desenvolvimento.....	35
Referências.....	36

Introdução

Este documento apresenta a documentação do trabalho desenvolvido para a disciplina de Programação Orientada a Objetos, do curso de Bacharelado em Sistemas de Informação, oferecida pelo Instituto Federal de Minas Gerais (IFMG) - Campus Ouro Branco. A atividade foi proposta pelo professor Saulo e consiste na implementação de um jogo de quebra-cabeça conhecido como "Racha Cuca" e o código desenvolvido está disponível no endereço <https://github.com/brazillucas/POO22024>.

O jogo "Racha Cuca" é composto por um tabuleiro de 3 linhas e 3 colunas, com 9 peças numeradas de 1 a 8 e um espaço em branco. O objetivo do jogador é organizar essas peças em ordem crescente (1, 2, 3, 4, 5, 6, 7, 8, espaço em branco) movendo as peças adjacentes ao espaço vazio até alcançar a sequência correta.

O código permite ao jogador escolher o nível de dificuldade para iniciar a partida: fácil, médio ou difícil. Cada nível possui uma quantidade máxima de embaralhamentos (20, 40 e 80, respectivamente), tornando o desafio progressivamente mais complexo. Quando o jogador completa a sequência correta, o jogo é finalizado e uma tela com o resultado é exibida, dando ao usuário a opção de sair ou iniciar uma nova partida. Da mesma forma, o jogador pode optar por sair ou recomeçar em qualquer momento.

Implementação

O programa foi desenvolvido visando ser o mais modular possível, assim sendo, possui diversos métodos para realizar as mais diversas funções.

Toda a explicação será feita considerando a disposição original dos métodos no código original.

scan

Variável responsável por **capturar a entrada do usuário** durante toda a execução do programa.

Objetivo:

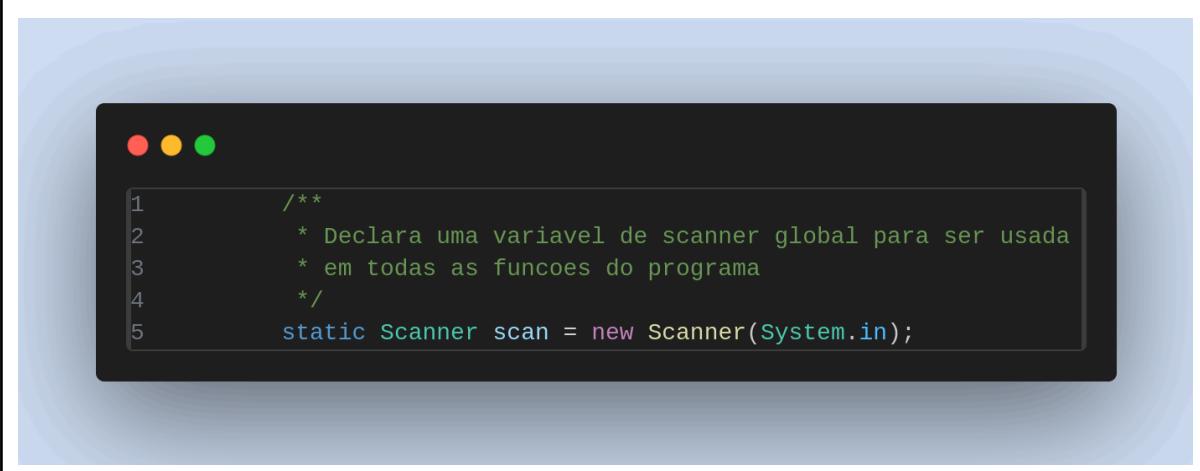
- **Centralizar a leitura de dados:** Evita a criação de múltiplas instâncias de **Scanner** em diferentes métodos, simplificando o código e reduzindo o risco de erros

relacionados ao fechamento do recurso.

- **Garantir a consistência da entrada:** Assegura que todas as leituras de dados sejam realizadas através de um único ponto de entrada, facilitando a manutenção e a depuração do código.

Detalhes da Implementação:

- **Visibilidade:** Declarada como `static` para garantir que exista apenas uma instância em todo o programa.
- **Escopo:** Inicializada no escopo da classe, fora de qualquer método, para que seja acessível por todos os métodos da classe.
- **Dificuldades de implementação:** nas primeiras versões do código, era declarado um scanner em cada função que deveria ler uma variável, mas isso gerava conflitos e erros de compilação que levaram a buscar uma nova forma de declaração essa variável em escopo global.



```
1      /**
2      * Declara uma variavel de scanner global para ser usada
3      * em todas as funcoes do programa
4      */
5      static Scanner scan = new Scanner(System.in);
```

Declaração da variável `scan`

Após declarar o scanner do jogo, começamos a implementar os métodos necessários para o fluxo do código.

Abaixo os principais são citados e explicados.

`encerraJogo()`

Finaliza o jogo chamando `System.exit(0)` para encerrar de forma correta o programa não deixando restos na memória.

```
1  /**
2   * Imprime uma mensagem de despedida e finaliza a execu
3   *      ao do programa
4   */
5   public static void encerraJogo() {
6       System.out.println(
7           "Obrigado por utilizar nosso programa!");
8       System.out.println("Programa encerrando...");
9       System.exit(0);
10 }
```

Código da função `encerraJogo()`

imprimeLinha()

Permite **criar uma linha para dividir partes da impressão**.

Objetivo:

- **Organização da saída:** Gera um ambiente mais organizado que não mistura as saídas.
- **Delimitação de impressões:** Oferece uma delimitação clara de cada saída do programa.

Detalhes da implementação:

- **Laço for:** Intera até imprimir a quantidade de caracteres escolhidos.
- **Saída:** Imprime uma linha na tela, utilizando o caractere “=”.

```
1  /**
2   * Imprime uma linha formada pelo caractere "=" para separar
3   * as mensagens exibidas ao usuario
4   * @param quantidade
5   * Quantas vezes deve ser impresso o caractere
6   * para formar a linha
7   */
8   public static void imprimeLinha(int quantidade) {
9       for (int contador = 0; contador <= quantidade; contador++)
10      {
11          System.out.print("=");
12      }
13      System.out.println("");
14 }
```

Código da função **imprimeLinha()**

criarTabuleiro()

Responsável por **inicializar o tabuleiro do jogo em seu estado inicial ordenado**.

Objetivo:

- **Preparar o jogo:** Cria a estrutura de dados que representa o tabuleiro, com todas as peças em suas posições iniciais.
- **Definir o estado inicial:** Estabelece um ponto de partida conhecido para o jogo, facilitando a implementação de outras funcionalidades, como a verificação de vitória e o embaralhamento.

Detalhes da Implementação:

- **Matriz 3x3:** Utiliza uma matriz bidimensional de tamanho 3x3 para representar o tabuleiro.
- **Preenchimento sequencial:** Preenche a matriz com os números de 1 a 8 de forma sequencial, deixando a última posição vazia (normalmente representada por 0).
- **Retorno:** Retorna a matriz inicializada, que será utilizada em outras partes do jogo.

```
1      /**
2       * Cria uma tabuleiro 3x3 inicializado com os
3       * números de 1 a 8, deixando a ultima posicao vazia.
4       * Os números são preenchidos sequencialmente,
5       * simulando um estado inicial ordenado.
6       *
7       * @return tabuleiro Uma array bidirecional representando
8       *         o tabuleiro o jogo.
9       */
10      public static int[][] criarTabuleiro() {
11          int[][] tabuleiro = new int[3][3];
12          int numero = 1;
13
14          for (int linha = 0; linha < 3; linha++) {
15              for (int coluna = 0; coluna < 3; coluna++) {
16                  if (numero < 9) {
17                      tabuleiro[linha][coluna] = numero;
18                      numero++;
19                  }
20              }
21          }
22
23          return tabuleiro;
24      }
```

Código da função **criarTabuleiro()**

imprimirTabuleiro()

Responsável por **exibir o estado atual do tabuleiro na tela de forma visualmente atrativa**.

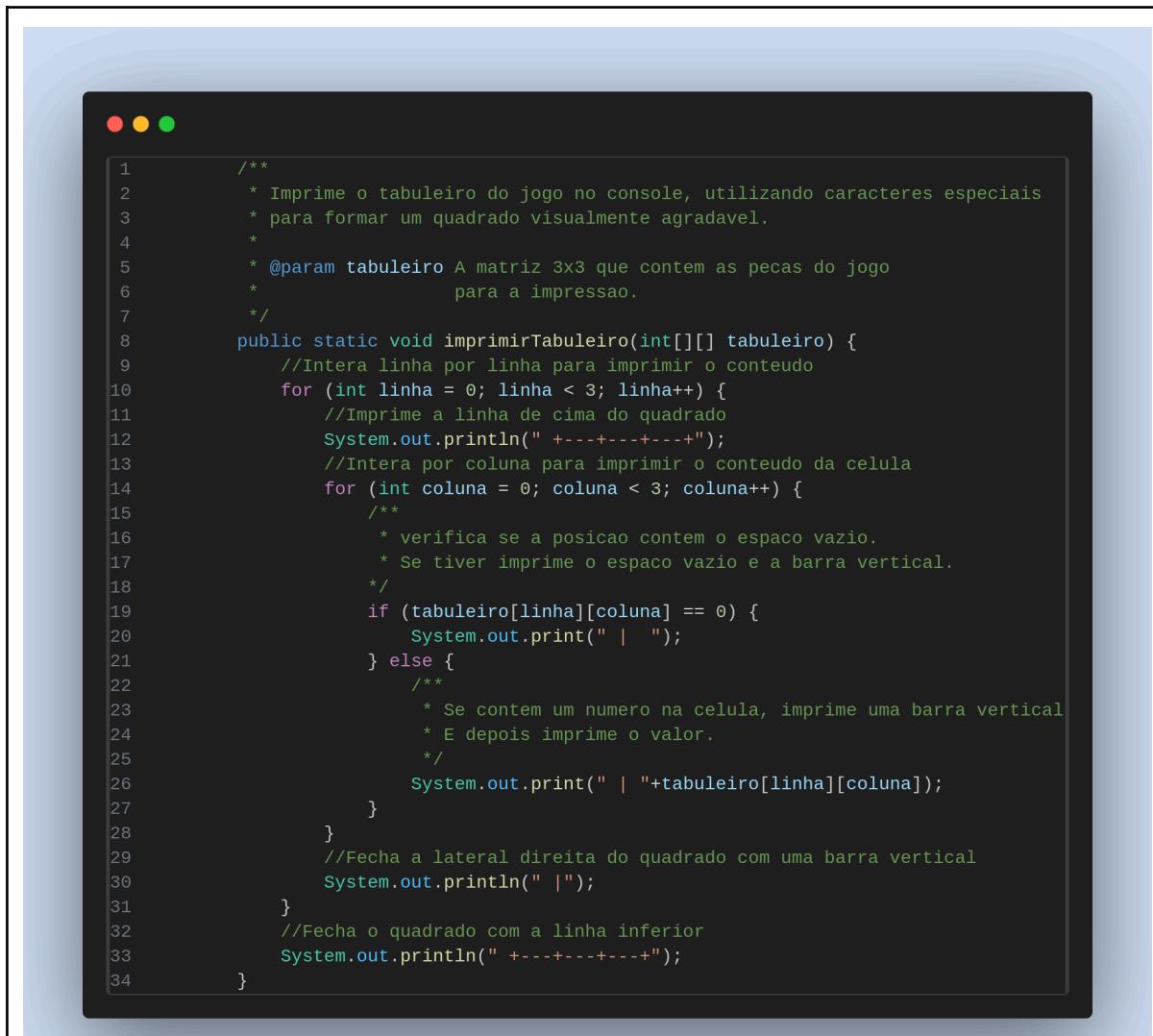
Objetivo:

- **Feedback visual:** Permite ao jogador visualizar o estado atual do jogo, facilitando a tomada de decisões.
- **Interface amigável:** Utiliza caracteres especiais e formatação para criar uma

representação visual clara e intuitiva do tabuleiro.

Detalhes da Implementação:

- **Iteração pela matriz:** Percorre cada elemento da matriz que representa o tabuleiro.
- **Formatação:** Utiliza caracteres especiais para delimitar as células do tabuleiro e criar uma visualização mais clara.
- **Mapeamento de valores:** Mapeia cada valor numérico da matriz para um caractere específico, representando a peça correspondente. Na prática, garante que quando achar a peça vazia imprima os caracteres correspondentes, assegurando o padrão da interface.
- **Saída:** Imprime o tabuleiro formatado na tela.



```
1  /**
2  * Imprime o tabuleiro do jogo no console, utilizando caracteres especiais
3  * para formar um quadrado visualmente agradável.
4  *
5  * @param tabuleiro A matriz 3x3 que contém as peças do jogo
6  *                   para a impressão.
7  */
8  public static void imprimirTabuleiro(int[][] tabuleiro) {
9      // Itera linha por linha para imprimir o conteúdo
10     for (int linha = 0; linha < 3; linha++) {
11         // Imprime a linha de cima do quadrado
12         System.out.println(" +---+---+---+");
13         // Itera por coluna para imprimir o conteúdo da célula
14         for (int coluna = 0; coluna < 3; coluna++) {
15             /**
16              * Verifica se a posição contém o espaço vazio.
17              * Se tiver imprime o espaço vazio e a barra vertical.
18              */
19             if (tabuleiro[linha][coluna] == 0) {
20                 System.out.print(" |   ");
21             } else {
22                 /**
23                  * Se contém um número na célula, imprime uma barra vertical
24                  * e depois imprime o valor.
25                  */
26                 System.out.print(" | " + tabuleiro[linha][coluna]);
27             }
28         }
29         // Fecha a lateral direita do quadrado com uma barra vertical
30         System.out.println(" | ");
31     }
32     // Fecha o quadrado com a linha inferior
33     System.out.println(" +---+---+---+");
34 }
```

Código do método **imprimirTabuleiro()**

trocaPecas()

Responsável por atualizar o estado do tabuleiro ao realizar uma movimentação. Recebe como parâmetros o tabuleiro atual, a linha e a coluna da peça que será trocada com o espaço vazio.

Objetivo:

- **Implementar a mecânica básica do jogo:** Ao trocar a posição de uma peça com o espaço vazio, permite que o jogador aproxime-se da configuração final do quebra-cabeça.
- **Facilitar a implementação de outras funcionalidades:** Serve como base para outras funcionalidades do jogo, como o embaralhamento inicial.

Detalhes da Implementação:

- **Validação:** Assume que a jogada já foi validada antes de chamar esta função, garantindo que a peça escolhida possa ser movida.
- **Troca de Posições:** Realiza a troca da peça selecionada com o espaço vazio no tabuleiro.
- **Informação:** Informa o jogador qual foi a peça alterada.
- **Retorno:** Retorna o tabuleiro atualizado após a troca.
- **Versão inicial:** Nas primeiras versões do código (*vide o repositório no github*) esse método recebia um valor booleano para indicar se a mudança estava sendo feita para um tabuleiro novo (embaralhando) ou para um movimento do jogador. Para que a legibilidade do código não fosse afetada e a capacidade de reuso de código não fosse prejudicada, foi necessário simplificar e o método para apenas realizar as mudanças.

Como o tabuleiro é uma estrutura complexa no Java, ele é passado por referência e não precisa ser retornado, a mudança é feita diretamente no tabuleiro original.

```
1      /**
2       * Troca duas pecas do tabuleiro de posicao.
3       * Troca a peça na posicao especificada por
4       * "linha" e "coluna" com o espaço vazio.
5       *
6       * @param tabuleiro A matriz 3x3 onde ira ocorrer a troca.
7       * @param linha A linha do valor que o usuario deseja trocar
8       *               de lugar.
9       * @param coluna A coluna do valor que o usuario deseja trocar
10      *               de lugar.
11
12     * @return O tabuleiro, agora com as duas pecas trocadas.
13    */
14    public static int[][] trocaPecas(int[][] tabuleiro, int linha, int coluna) {
15        //Acha a linha e a coluna da celula vazia
16        int linha1 = 0;
17        int coluna1 = 0;
18
19        //Procura a posicao da celula vazia
20        for (int i = 0; i < 3; i++) {
21            for (int j = 0; j < 3; j++) {
22                if (tabuleiro[i][j] == 0) {
23                    linha1 = i;
24                    coluna1 = j;
25                    break;
26                }
27            }
28        }
29
30        System.out.println("Peca que mudou: " + tabuleiro [linha][coluna]);
31
32        //Recebe temporariamente o numero que sera trocado
33        int temporario = tabuleiro[linha][coluna];
34
35        //faz a troca dos dois numeros
36        tabuleiro[linha][coluna] = tabuleiro[linha1][coluna1];
37        tabuleiro[linha1][coluna1] = temporario;
38
39        //Retorna o tabuleiro com 2 pecas com as posicoes trocas
40        return tabuleiro;
41    }
```

Código da função **trocaPecas()**

embaralharTabuleiro()

Embaralha o tabuleiro do jogo, gerando uma configuração aleatória a partir de um estado inicial. A dificuldade do embaralhamento é definida pelo usuário e influencia o número de trocas realizadas entre as peças.

Objetivos:

- **Aleatoriedade:** Garantir que cada partida tenha um início diferente, aumentando a rejogabilidade.
- **Dificuldade:** Permitir que o jogador escolha a dificuldade do jogo, ajustando o nível de desafio.
- **Validação:** Assegura que as trocas entre as peças sejam válidas, respeitando as regras do jogo.

Detalhes da Implementação:

- **Dificuldade:** A função `selecionarDificuldade()` determina o número de trocas a serem realizadas.
- **Aleatoriedade:** Um objeto `Random` é utilizado para gerar números aleatórios para as linhas e colunas das peças a serem trocadas.
- **Validação:** A função `validarJogada()` verifica se a troca proposta é válida de acordo com as regras do jogo.
- **Loop de embaralhamento:** Um loop `do-while` garante que o tabuleiro seja embaralhado até que esteja em um estado aleatório e não ordenado.
- **Troca de peças:** A função `trocaPecas()` realiza a troca entre a peça selecionada aleatoriamente e a peça vazia.
- **Verificação de repetições:** A função verifica se a mesma peça não está sendo trocada consecutivamente.
- **Condição de parada:** O loop termina quando, depois da quantidade de trocas determinadas pela dificuldade escolhida, o tabuleiro não está ordenado, conforme retornado pela função `conferirTabuleiro()`.

Dificuldades da Implementação:

A função de embaralhamento foi de longe a mais difícil de implementar. Pois, teve que ser refatorada diversas vezes para que o funcionamento corresponesse à todos os requisitos exigidos.

A lógica para definir que dois números aleatórios que devem corresponder às peças adjacentes à peça vazia demorou um tempo considerável para ser finalizada. Inicialmente estava inclusa na função `trocaPecas()`, mas isso deixava o código daquela função mais complexo do que era necessário. Então foi desenvolvido dentro do embaralhamento.

Juntamente com o requisito de trocar apenas peças que estão ao lado de zero, surgiu a necessidade de garantir que não iria ocorrer uma troca segundo o exemplo: primeiro 0-8 e

depois 8-0. Essa implementação era importante para garantir que o jogo não estava trocando repentinamente apenas uma peça de lugar.

```
1      /**
2       * Embaralha o tabuleiro criado no inicio de cada partida. Com o
3       * auxilio da funcao selecionarDificuldade() define quantas trocas
4       * serao feitas no tabuleiro. Com a dificuldade selecionada, gera 2
5       * numeros para escolher uma celula que sera trocada com a celula
6       * vazia. Manda essa celula para o metodo validarJogada() e se for
7       * entra no loop para chamar a funcao trocaPecas() para efetuar
8       * as trocas quantas vezes forem necessarias.
9       * Enquanto o tabuleiro finalizar o loop for ordenado, embaralha
10      * novamente.
11      *
12      * @param tabuleiro Uma matriz 3x3 que contem os numeros que devem ser
13      *                   trocados na execucao da funcao.
14      * @return O tabuleiro embaralhado conforme a dificuldade escolhida.
15      */
16     public static int[][] embaralharTabuleiro(int[][] tabuleiro) {
17         /*Chama a funcao selecionarDificuldade() para definir quantas
18          * trocas devem ser realizadas no tabuleiro
19          */
20
21
22         int dificuldade = selecionarDificuldade();
23
24         //Instancia um gerador de numero aleatorio
25         Random aleatorio = new Random();
26
27         int linhaMudanca = -1;
28         int colunaMudanca = -1;
29         boolean podeMudar = false;
30         boolean embaralhou = true;
31         int ultimaMudanca = -1;
32
33         // Embaralha o tabuleiro ate que ele nao retorne ordenado
34         do {
35             //Chama a funcao trocaPecas() para efetivar o
36             //embalhamento do tabuleiro
37             for (int vezes = 0; vezes < dificuldade;vezes++) {
38
39                 // Gera dois numeros aleatorios para selecionar a celula a ser trocada
40                 // e valida se a jogada pode ser feita. Se nao puder, gera novos numeros.
41                 while(podeMudar == false) {
42                     linhaMudanca = aleatorio.nextInt(3);
43                     colunaMudanca = aleatorio.nextInt(3);
44
45                     // Confirma se o movimento pode ser feito
46                     podeMudar = validarJogada(linhaMudanca, colunaMudanca, tabuleiro);
47
48                     /**
49                      * Verifica se a celula escolhida para mudar nao e a mesma que foi
50                      * mudada na ultima iteracao.
51                      * Se for diferente, salva o valor da peca que mudou de posicao
52                      * dentro da variavel ultimaMudanca. Senao, continua sorteando
53                      * numeros aleatorios.
54                     */
55
56                     if (tabuleiro[linhaMudanca][colunaMudanca] == ultimaMudanca) {
57                         podeMudar = false;
58                     } else if (podeMudar == true) {
59                         ultimaMudanca = tabuleiro[linhaMudanca][colunaMudanca];
60                     }
61                 }
62                 podeMudar = false;
63
64                 tabuleiro = trocaPecas(tabuleiro, linhaMudanca, colunaMudanca);
65             }
66             embaralhou = conferirTabuleiro(tabuleiro);
67             //Verifica se o tabuleiro continua ordenado, se estiver, embaralha novamente
68         } while (embaralhou == true);
69
70         //Retorna o tabuleiro embaralhado
71         return tabuleiro;
72     }
```

Código do método embaralharTabuleiro()

menu()

Responsável por **apresentar as opções disponíveis ao jogador e capturar sua escolha**.

Objetivo:

- **Interação com o usuário:** Permite que o jogador escolha entre as diferentes ações disponíveis no jogo (iniciar nova partida, ler as instruções ou sair do jogo).
- **Validação de entrada:** Garante que a opção escolhida pelo jogador seja válida, evitando erros e interrupções inesperadas no jogo.

Detalhes da Implementação:

- **Apresentação das opções:** Exibe um menu na tela, listando as opções disponíveis para o jogador.
- **Leitura da escolha:** Utiliza a função [lerOpcao\(\)](#) para capturar a opção escolhida pelo usuário.
- **Validação:** Verifica se a opção escolhida é válida e, caso contrário, exibe uma mensagem de erro e apresenta novamente o menu.
- **Retorno:** Retorna a opção escolhida pelo jogador, que será utilizada para controlar o fluxo do jogo.

Um conceito interessante utilizado para a construção do menu é a recursividade, que nada mais é do que a chamada da função dentro dela mesma. Assim, não é necessário criar um loop na chamada do menu, já que todo o jogo passa por ele e uma vez não sendo escolhida uma opção válida, o menu é chamado novamente.

```
1      /**
2       * Imprime um pequeno menu para apresentar ao usuario
3       * as opcoes do sistema.
4       * Conforme a escolha, uma funcao especifica eh chamada.
5       * Caso nenhuma opcao valida seja escolhida, uma mensagem
6       * de erro eh exibida e a funcao menu() eh chamada
7       * novamente. Evitando a necessidade de um loop
8       * na chamada da funcao, ao usar recursividade.
9       */
10      public static void menu() {
11
12          imprimeLinha(25);
13          System.out.println("Escolha das opcoes abaixo: ");
14          System.out.println("0 - Sair");
15          System.out.println("1 - Comecar um novo jogo");
16          System.out.println("2 - Instrucoes");
17          imprimeLinha(25);
18
19          System.out.print("Opcao Selecionada: ");
20          /* Cria uma variavel que sera usada no Switch abaixo
21           * Essa variavel e iniciada recebendo o retorno da
22           * funcao lerOpcao(); que retorna um numero
23           */
24          int opcao = lerOpcao();
25
26          switch(opcao){
27              case 0:
28                  /**
29                   * Caso o usuario queira encerrar o programa, chama
30                   * a funcao encerraJogo() que ira finalizar o jogo
31                   */
32                  encerraJogo();
33                  break;
34              case 1:
35                  /**
36                   * Caso o usuario queira jogar uma partida, chama
37                   * a funcao comecarJogo() que ira executar a logica
38                   * do jogo
39                   */
40                  comeclarJogo();
41                  break;
42              case 2:
43                  /**
44                   * Caso o usuario queira ler as instrucoes, chama
45                   * a funcao mostrarInstrucoes() que ira listar as
46                   * instrucoes necessarias para jogar
47                   */
48                  mostrarInstrucoes();
49                  break;
50              default:
51                  /**
52                   * Em caso de nao usar uma das opcoes validas, a
53                   * tela e limpa e um aviso aparece para o usuario
54                   * logo em seguida eh chamado o menu() novamente,
55                   * assim o jogador pode continuar tentando
56                   * acertar ou sair do jogo.
57                   */
58                  limparConsole();
59                  System.out.println("Opcao nao encontrada!");
60                  System.out.println("Reinsira uma opcao valida.");
61                  menu();
62                  break;
63          }
64      }
```

Código do método menu()

IerOpcao()

Responsável por receber as escolhas realizadas pelo usuário durante a utilização do jogo.

Objetivo:

- **Receber opção do usuário:** Permite que o usuário insira o número correspondente a sua opção.

Detalhes da implementação:

- **Validação de entrada:** Garante que o usuário insira um número inteiro e somente isso.
- **Tratamento de erros:** utilizando uma estrutura **try() catch()** para verificar a entrada, o programa garante que caso seja inserido um caractere especial ou uma letra seja possível continuar a rodar.
- **Limpeza de buffer:** Limpa a entrada depois de ler o inteiro, assegurando que não ocorreram bugs na próxima utilização do scan.

Este método foi particularmente difícil de desenvolver por utilizar conceitos ainda não passados em aula, mas que eram de extrema importância para a criação de um código estável.

Nas primeiras versões essa verificação com **try catch** não existia e foi notado o encerramento precoce do jogo por um deslize qualquer na hora de digitar a opção desejada. Para contornar este bug, foi escolhida a solução embutida na própria linguagem que oferece a opção de tratar erros.

```
1      /**
2       * Le a entrada de uma variavel inteira que o usuario
3       * escolheu.
4       *
5       * @return
6       * Um valor inteiro representando a escolha do usuario.
7       */
8       public static int lerOpcao() {
9           int opc = -1;
10          boolean entradaValida = false;
11          /**
12           * Le a opcao selecionada ate obter uma
13           * entrada no formato correto
14           */
15          while (!entradaValida) {
16              try {
17                  opc = scan.nextInt();
18                  limparBuffer();
19                  entradaValida = true;
20              } catch (Exception e) {
21                  System.out.print(
22                      "Entrada invalida!! Digite um numero inteiro: ");
23                  limparBuffer();
24              }
25          }
26      return opc;
27 }
```

Código da função lerOpcao()

selecionarDificuldade()

Dedicada a receber a dificuldade escolhida pelo usuário.

Objetivo:

- **Exibir as opções:** Exibe as dificuldades possíveis de selecionar.
- **Mudanças de peças:** Entrega a quantidade de mudanças que serão feitas na inicialização do jogo.

Detalhes da implementação:

- **Seleção garantida:** Uma vez iniciada, a função só retorna para o fluxo geral do jogo se uma dificuldade for escolhida.
- **Entrada validada:** Como forma de aceitar apenas os dados corretos, a leitura acontece chamando a função [lerOpcao\(\)](#).
- **Retorno:** um inteiro que representa a quantidade de vezes que o tabuleiro deverá ser embaralhado.



```
1  /**
2   * Exibe um menu com as dificuldades disponiveis.
3   * Le a escolha do usuario e baseado nessa escolha
4   * gera um retorno especifico.
5   * Se a escolha nao for valida, o programa continua
6   * aguardando em loop uma resposta correta.
7   *
8   * @return 20, 40 ou 80, que sera usado para definir quantas trocas
9   *         de posicoes das pecas serao efetuadas.
10  */
11 public static int selecionarDificuldade() {
12
13     limparConsole();
14     imprimeLinha(10);
15     System.out.println("Niveis de dificuldade:");
16     System.out.println("1 - Facil");
17     System.out.println("2 - Medio");
18     System.out.println("3 - Dificil");
19     imprimeLinha(10);
20     System.out.println("Selecione a dificuldade do jogo.");
21
22     while (true) {
23         System.out.print("Opcão Selecionada: ");
24         int dificuldade = lerOpcao();
25
26         switch (dificuldade) {
27             case 1:
28                 return 20;
29             case 2:
30                 return 40;
31             case 3:
32                 return 80;
33             default:
34                 System.out.println("Opcão invalida!");
35         }
36     }
37 }
38 }
```

Código da função `selecionarDificuldade()`

moverPeca()

Confere se tudo está alinhado para chamar a função [trocaPecas\(\)](#).

Objetivos:

- **Receber a entrada do jogador:** Conferir se o movimento está de acordo com as regras do jogo.
- **Atualizar o estado do jogo:** Modificar o tabuleiro para refletir o movimento realizado.
- **Confirmar se o tabuleiro foi alterado:** a cada rodada a função confirma se o tabuleiro foi ou não modificado.
- **Manter a clareza das saídas:** a cada mudança finalizada com sucesso o terminal é limpo e imprime novamente o tabuleiro, desta vez atualizado.

Detalhes da implementação:

- **Não permitir mudanças de peças inexistentes:** Verifica se a peça selecionada pelo jogador é menor que 1 ou maior que 8, pois as peças do tabuleiro variam nessa faixa. Também confere se é igual a 9, pois esse é o comando de saída.
- **Validar a jogada:** Cada vez que a peça que o usuário deseja mudar é encontrada dentro do tabuleiro, as coordenadas da peça são enviadas para a função [validarJogada\(\)](#) que retorna informando se aquela jogada está dentro das regras do jogo.
- **Troca de peças:** Uma vez validada a jogada, a função chama a [trocaPecas\(\)](#) para alterar o tabuleiro.
- **Retorno:** Um booleano, que retorna positivo caso a mudança tenha ocorrido e falso se algum problema for encontrado.

```
1      /**
2       * Move a peca no tabuleiro (uma matriz 3x3) de lugar
3       * desde que o seu valor seja valido (entre 1 e 8) e esteja
4       * ao lado da peca vazia.
5       *
6       * @param tabuleiro A matriz 3x3 que contem os valores a
7       *                   serem trocados.
8       * @param jogada O valor da peca que o usuario deseja mover
9       *                   para o espaco vazio.
10      * @return Verdadeiro ou falso, a depender se a peca foi movida ou nao.
11      */
12     public static boolean moverPeca(int[][] tabuleiro, int jogada) {
13
14         /**
15          * Verifica se o valor a ser trocado nao eh maior
16          * que o valor maximo presente no tabuleiro (8) ou
17          * menor que o valor minimo (0)
18          */
19         if (jogada < 1 || jogada > 9) {
20             limparConsole();
21             System.out.println("Peca incorreta, tente novamente!");
22             imprimirTabuleiro(tabuleiro);
23             return false;
24         }
25         if (jogada == 9) {
26             return false;
27         }
28
29         for (int linha = 0; linha < 3; linha++) {
30             for (int coluna = 0; coluna < 3; coluna++) {
31
32                 if (tabuleiro[linha][coluna] == jogada) {
33                     /**
34                      * Valida se o movimento pode ser executado com a peca
35                      * na posicao atual, considerando as regras estabelecidas
36                      * em validaJogada()
37                      */
38                     if (validarJogada(linha, coluna, tabuleiro) == true) {
39                         limparConsole();
40                         tabuleiro = trocaPecas(tabuleiro, linha, coluna);
41                         imprimirTabuleiro(tabuleiro);
42                         return true;
43                     }
44                 /**
45                   * Caso a jogada nao seja valida, limpa a tela
46                   * E informa o usuario que nao pode mudar aquela peca
47                   */
48                 limparConsole();
49                 System.out.println("Movimento invalido, tente novamente!");
50                 imprimirTabuleiro(tabuleiro);
51                 return false;
52             }
53         }
54     }
55 }
56 /**
57  * Depois de cada execucao, imprime o tabuleiro para o
58  * jogador saber qual o estado atual do jogo
59  */
60 return false;
61 }
```

Código da função moverPeca()

validarJogada()

Verifica se a jogada com a peça escolhida pode ser realizada seguindo as regras do jogo.

Objetivo:

- **Prevenir movimentos inválidos:** Não permite que o jogador tente mudar uma peça que não está do lado de um espaço vazio.

Detalhes da Implementação:

- **Verificar qual a posição da peça:** Se a peça estiver nas bordas do tabuleiro, a função procura a célula vazia apenas nas partes internas.
 - Para células que estejam na linha 2, verifica na linha anterior pela célula vazia;
 - Para células que estejam na linha 1, verifica nas linhas anterior e posterior pela célula vazia;
 - Para células que estejam na linha 0, verifica na linha seguinte pela célula vazia.
 - A lógica de verificação nas colunas é a mesma.
- **Modularidade:** Captura o tamanho do tabuleiro com o método `.length`, tornando o código utilitário em tabuleiros de tamanhos diferentes.
- **Retorno:** Retorna `True` se a peça vazia estiver ao lado da peça escolhida, ou `False` se não estiver.

Essa função foi particularmente difícil de finalizar, pois inicialmente ela foi feita de maneira muito complexa, o que atrapalhava a legibilidade do código. Isso porque ela fazia todas as verificações da imagem abaixo, uma por uma (com 9 `IFs` diferentes).

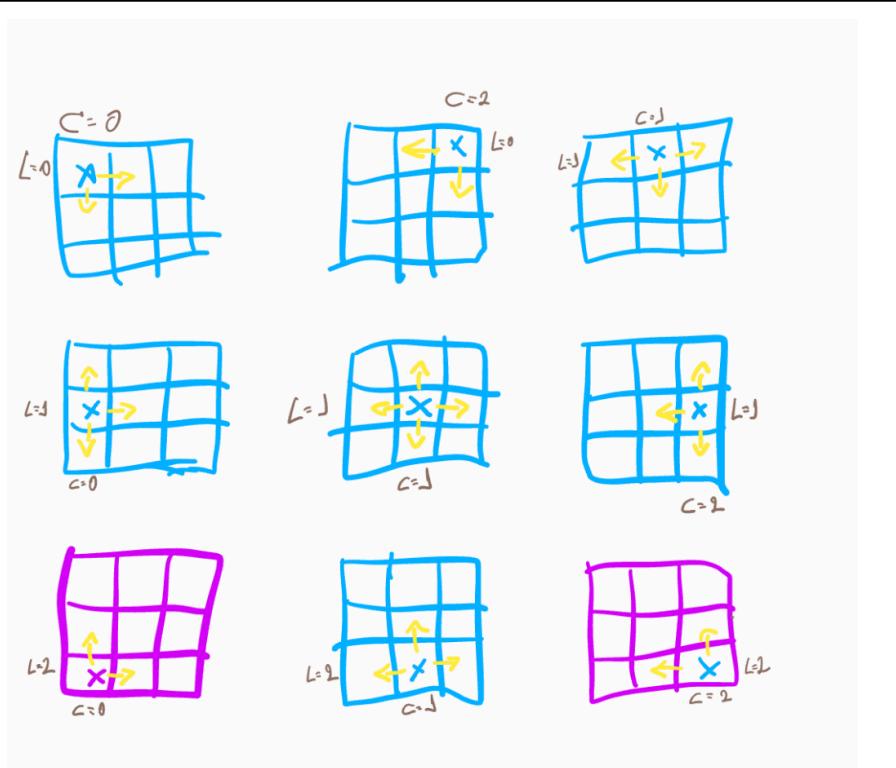


Diagrama criado para auxiliar no desenvolvimento da lógica de validação.

Gerada no aplicativo de notas Google Keep.

Depois de algumas tentativas, foi escolhido utilizar uma versão que verificava apenas 3 condições das linhas e 3 condições das colunas. Caso nenhuma dessas condições fosse verdadeira, assume-se que a peça não está ao lado da célula vazia.

```
1      /**
2       * Verifica se a posicao da jogada e valida, ou seja,
3       * se a peca vazia esta ao lado da peca que o jogador
4       * deseja mover.
5       * Considera tambem as posicoes de borda do tabuleiro.
6       *
7       * @param linha A linha da posicao do valor a ser mudado
8       * @param coluna A coluna da posicao do valor a ser mudado
9       * @param tabuleiro A matriz com os valores onde a funcao ira
10      *                   procurar a celula vazia para fazer as comparacoes
11      * @return Falso ou verdadeiro, para saber se a mudanca e valida ou nao
12      */
13     public static boolean validarJogada(int linha, int coluna, int[][]tabuleiro) {
14
15         // Verifica se a posicao de cima e valida e esta vazia
16         if (linha > 0 && tabuleiro[linha-1][coluna] == 0) {
17             return true;
18         }
19
20         // Verifica se a posicao de baixo e valida e esta vazia
21         if (linha < 3 && tabuleiro[linha+1][coluna] == 0) {
22             return true;
23         }
24
25         // Verifica se a posicao a esquerda e valida e esta vazia
26         if (coluna > 0 && tabuleiro[linha][coluna -1] == 0) {
27             return true;
28         }
29
30         // Verifica se a posicao a direira e valida e esta vazia
31         if (coluna < 3 && tabuleiro[linha][coluna + 1] == 0) {
32             return true;
33         }
34
35         //Se nenhuma das condicoes acima for verdadeira, retorna falso
36         return false;
37     }
38 }
```

Código do método **validarJogada()**

conferirTabuleiro()

Verifica se o tabuleiro está ordenado.

Objetivo:

- **Validar a ordenação do tabuleiro:** Confere se as peças do tabuleiro estão organizadas de 1 a 8, com a peça vazia no final.

Detalhes da Implementação:

- **Vetor de 9 posições:** Cria um array e preenche com os 8 valores das peças ordenados e um 0 na última posição.
- **Intera pela matriz:** A cada posição da matriz, compara seu valor com uma posição

posição do vetor e, caso seja igual, incrementa a posição do vetor para a próxima interação.

- **Retorno:** Se em alguma comparação não for igual, retorna **False**, indicando que o tabuleiro não está ordenado. Se tudo estiver certo, retorna **True** no fim do método.

Esse método será utilizado algumas vezes durante o jogo para garantir que certas ações sejam realizadas ou não a depender do status de ordenação do tabuleiro.

```
1      /**
2       * Gera um vetor ordenado para conferir se o tabuleiro esta na ordem
3       * correta (1 a 8) e depois compara com o tabuleiro para verificar
4       * se tambem esta ordenado.
5       *
6       * @param tabuleiro A matrix 3x3 usada como tabuleiro de jogo e que aqui
7       *                   sera verificada para saber se o jogador ja conseguiu.
8       *
9       * @return Um valor booleano para indicar se o tabuleiro o tabuleiro
10      *        ja esta ordenado.
11     */
12    public static boolean conferirTabuleiro(int[][] tabuleiro) {
13        int tabuleiroCorrigido[] = new int[9];
14
15        for (int i = 0; i < 8; i++) {
16            tabuleiroCorrigido[i] = i+1;
17        }
18
19        tabuleiroCorrigido[8] = 0;
20
21        int contador = 0;
22
23        for (int linha = 0; linha < 3; linha++) {
24            for (int coluna = 0; coluna < 3; coluna++) {
25                if(tabuleiro[linha][coluna] != tabuleiroCorrigido[contador]) {
26                    return false;
27                }
28                contador++;
29            }
30        }
31        return true;
32    }
```

Código da função conferirTabuleiro()

limparBuffer()

Assegura que nenhuma entrada será prejudicada por restos de leituras anteriores.

Objetivo:

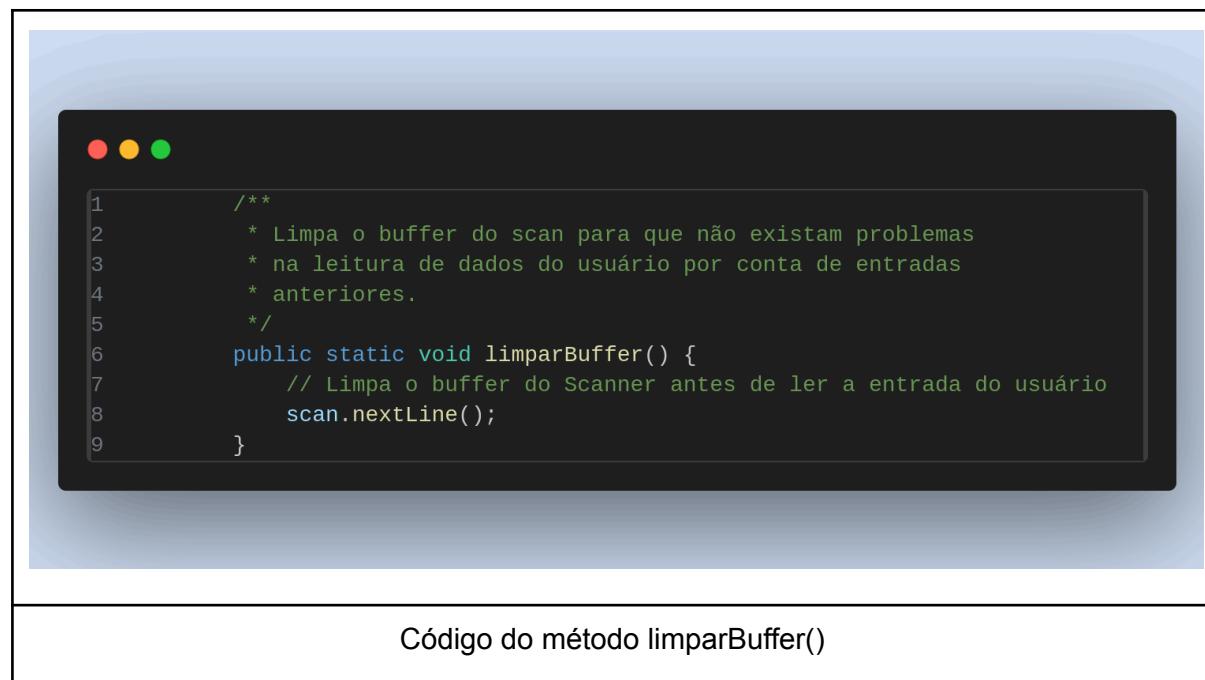
- **Leituras limpas:** Consome qualquer possível “\n” restante de algum uso do [scan](#).

Detalhes da Implementação:

- **.nextLine:** Utiliza o método `scan.nextLine()` que espera, nesse caso, um ENTER (“\n”) para continuar o programa. Caso exista algum travado no buffer da variável `scan`, ele é utilizado para dar essa continuidade para o `.nextLine`.

Essa função é utilizada toda vez que um `scan.nextInt()` é realizado, pois foi notado que após a leitura de um inteiro, toda vez que um `scan.nextLine()` fosse usado, havia um \n pendente que atrapalhava todo o fluxo de leituras do programa, muitas vezes dando comandos “fantasma” no meio de uma partida. Para contornar esse problema, o `.nextLine()` é utilizado.

Outro uso é para criar uma espécie de “timeout”, esperando que o usuário pressione ENTER antes de limpar a tela e seguir para um próximo fluxo (vide [encerrarPartida\(\)](#) e [continuarJogo\(\)](#)).



```
1  /**
2   * Limpa o buffer do scan para que não existam problemas
3   * na leitura de dados do usuário por conta de entradas
4   * anteriores.
5   */
6  public static void limparBuffer() {
7      // Limpa o buffer do Scanner antes de ler a entrada do usuário
8      scan.nextLine();
9  }
```

Código do método `limparBuffer()`

encerrarPartida()

Finalizar a partida exibindo o tabuleiro que o jogador ordenou e uma mensagem de parabéns para o jogador.

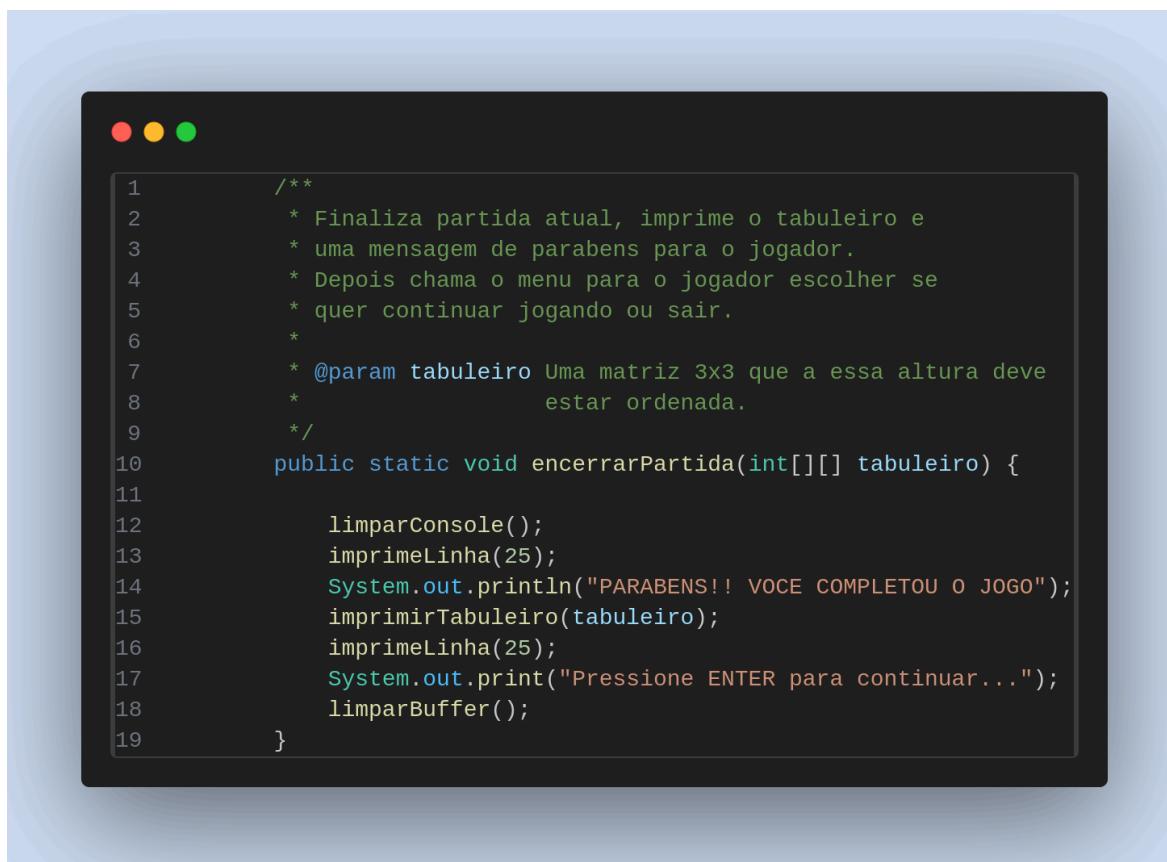
Objetivo:

- **Manter a organização da saída:** Trazer uma nova tela para exibir as mensagens.
- **Feedback do resultado:** Exibe para o jogador o tabuleiro no seu estado atual (ordenado).

Detalhes da Implementação:

- **Espera confirmação:** Após limpar a tela com o método `limparConsole()`, exibe o tabuleiro ordenado e aguarda o usuário pressionar ENTER para voltar para o `menu()`.

Essa é uma função que utiliza o `limpaBuffer()` de maneira diferente das funções de leitura. Aqui o método é utilizado para aguardar o usuário pressionar ENTER para continuar. Assim não precisa ser implementado nenhum método mais complexo para as telas de exibição nas quais o usuário não precisa de fato interagir escolhendo alguma opção.



```
1      /**
2       * Finaliza partida atual, imprime o tabuleiro e
3       * uma mensagem de parabens para o jogador.
4       * Depois chama o menu para o jogador escolher se
5       * quer continuar jogando ou sair.
6       *
7       * @param tabuleiro Uma matriz 3x3 que a essa altura deve
8       *                   estar ordenada.
9       */
10      public static void encerrarPartida(int[][] tabuleiro) {
11
12          limparConsole();
13          imprimeLinha(25);
14          System.out.println("PARABENS!! VOCE COMPLETOU O JOGO");
15          imprimirTabuleiro(tabuleiro);
16          imprimeLinha(25);
17          System.out.print("Pressione ENTER para continuar...");
18          limparBuffer();
19      }

```

Código do método `encerrarPartida()`

`comecarJogo()`

Responsável por **iniciar e gerenciar o fluxo principal do jogo**. Coordena diversas etapas, desde a criação e embaralhamento do tabuleiro até a interação com o jogador e a verificação da vitória.

Objetivos:

- **Iniciar o jogo:** Cria um novo tabuleiro, embaralha as peças e apresenta o tabuleiro inicial ao jogador.
- **Gerenciar o loop principal:** Controla a sequência de ações do jogador, desde a seleção de uma peça até a verificação da vitória.
- **Verificar condições de vitória:** Avalia se o jogador conseguiu organizar todas as peças no tabuleiro de acordo com o objetivo do jogo.
- **Controlar o fluxo do jogo:** Permite que o jogador continue jogando ou reinicie uma nova partida.

Detalhes da Implementação:

- **Criação e embaralhamento do tabuleiro:** Utiliza `criarTabuleiro()` e `embaralharTabuleiro()` para criar uma matriz representando o tabuleiro e embaralhá-la aleatoriamente.
- **Loop principal:** Emprega um loop `while` para repetir o processo de leitura da jogada do usuário, atualização do tabuleiro e verificação das condições de vitória até que o jogo seja encerrado.
- **Verificação de vitória:** Utiliza `conferirTabuleiro()` para comparar o estado atual do tabuleiro com o estado final desejado.
- **Contagem de jogadas:** Mantém um contador para registrar o número de movimentos realizados pelo jogador.
- **Interação com o usuário:** Emprega a função `lerOpcao()` para obter a entrada do usuário e a função `imprimirTabuleiro()` para exibir o estado atual do tabuleiro.
- **Mecanismo de pausa:** A cada 20 jogadas, o fluxo do jogo é interrompido para que o jogador possa decidir se deseja continuar. A função `continuarJogo()` apresenta uma mensagem ao usuário com as opções de continuar ou sair. A decisão do jogador determina se o loop principal do jogo continua ou é encerrado.
- **Verificação de saída:** A função `fecharPartida()` é chamada sempre que o usuário digita 9, para confirmar se ele realmente deseja sair do jogo. Caso o usuário

confirme a saída, a função `menu()` é chamada e o loop principal é encerrado implicitamente.

Dificuldades da implementação:

A implementação de um contador para as jogadas iniciou um desafio no desenvolvimento, pois o incremento deste contador só acontece se a jogada for de fato realizada (evitando que sejam contabilizadas as tentativas inválidas de mover peças).

No entanto, isso implica entrar num loop infinito ao entrar na validação de 20 jogadas. Quando o usuário diz que quer continuar e novamente coloca uma peça inválida para a jogada, entra-se novamente dentro da verificação.

A solução encontrada foi incrementar a variável depois da resposta do jogador, assim é garantido que não entrar em loop infinito.

O problema resultante dessa solução é que haverá sempre uma jogada a menos para a próxima verificação depois dos primeiros 20 movimentos.

Em resumo, a função `comecarJogo()` é como o botão "Iniciar" de um jogo de quebra-cabeça. Quando você clica nesse botão, o jogo é preparado para você. As peças são embaralhadas e o tabuleiro é mostrado na tela. A partir daí, você pode mover as peças tentando organizá-las da forma correta. A função fica responsável por acompanhar suas jogadas, verificar se você ganhou e te dar a opção de continuar jogando ou começar uma nova partida.

```
1      /**
2      * Cria um tabuleiro, embaralha ele e começa o jogo.
3      * A cada jogada, verifica se o tabuleiro está ordenado
4      * e se estiver, chama a função encerraPartida().
5      * A cada 20 jogadas, pergunta se o jogador quer continuar
6      * jogando ou se deseja começar um novo jogo.
7      */
8
9     public static void comecarJogo() {
10        int[][] tabuleiro = criarTabuleiro();
11        tabuleiro = embaralharTabuleiro(tabuleiro);
12
13        limparConsole();
14        imprimirTabuleiro(tabuleiro);
15
16        System.out.print("Selecione a peça que quer mover (Caso queira sair, digite 9): ");
17        int jogada = lerOpcao();
18        jogada = fecharPartida(jogada);
19        int jogadas = 0;
20
21
22        /** intera enquanto o jogador informar uma peça pra mexer, quando informar
23         * o código de sair, o loop Finaliza
24         */
25        while(true) {
26            boolean mudouPecas = moverPeca(tabuleiro, jogada);
27
28            /**
29             * A cada tentativa de mudança de peça o programa verifica se já
30             * está com a matriz ordenada, se estiver, chama o método
31             * encerraPartida().
32             */
33            if(conferirTabuleiro(tabuleiro)) {
34                encerrarPartida(tabuleiro);
35                break;
36
37            } else {
38                if (mudouPecas == true) {
39                    jogadas++;
40                }
41
42                /** A cada 20 jogadas, o programa pergunta se o jogador quer
43                 * parar ali e começar um novo jogo
44                 */
45                if (jogadas > 19 && (jogadas % 20) == 0) {
46
47                    limparConsole();
48                    imprimirTabuleiro(tabuleiro);
49                    System.out.println("Jogadas: " + jogadas);
50                    boolean continuar = continuarJogo();
51
52                    if (continuar == true) {
53                        limparConsole();
54                        imprimirTabuleiro(tabuleiro);
55                        jogadas++;
56                    } else {
57                        jogadas++;
58                        break;
59                    }
60                }
61
62                System.out.println("Jogadas: " + jogadas);
63                System.out.print("Selecione a peça que quer mover (caso queira sair, digite 9): ");
64                jogada = lerOpcao();
65
66                jogada = fecharPartida(jogada);
67            }
68        }
69        limparConsole();
70        menu();
71    }
72}
```

Código do método encerraPartida().

fecharPartida()

Verifica se o usuário deseja sair da partida.

Objetivos:

- **Impedir uma saída indesejada:** Confirma se o usuário quer realmente sair ou deseja continuar jogando, caso tenha sido um engano na escolha.

Detalhes da implementação:

- **Retorno:** Retorna o mesmo valor de `jogada` a menos que o usuário decida sair do jogo, nesse caso, a função `menu()` é chamada e o fluxo do programa é alterado.

Para evitar mais comparações, é utilizado o método `.toUpperCase()` do `scan` que transforma os dados digitados pelo usuário em letras maiúsculas.

Assim, apenas uma comparação é feita e caso seja confirmada a saída, o jogo chama o `menu()` e sai do fluxo principal do programa.

```
1      /**
2       * Recebe uma jogada e verifica se ela é igual a 9 (comando de saída).
3       * Se for, pergunta se o jogador deseja sair do jogo
4       * ou continuar jogando.
5       * @param jogada O último número escolhido pelo jogador.
6       * @return jogada O número inteiro escolhido pelo jogador antes da chamada
7           do método.
8      */
9      public static int fecharPartida(int jogada) {
10         if (jogada == 9) {
11             System.out.print("Tem certeza que deseja sair? (S - Sim ou Qualquer Tecla - Não) ");
12             String escolha = scan.nextLine().toUpperCase();
13
14             if (escolha.equals("S")) {
15                 limparConsole();
16                 menu();
17             }
18         }
19         return jogada;
20     }
```

Código do método `fecharPartida()`

continuarJogo()

Permite ao jogador decidir se quer continuar a partida atual ou iniciar uma nova.

Objetivos:

- **Obter a escolha do jogador:** Permite a escolha entre continuar a partida ou iniciar um novo jogo.

Detalhes da Implementação:

- **Loop infinito:** Utiliza um loop `while(true)` para garantir que a pergunta seja feita até que o jogador forneça uma resposta válida.
- **Leitura da escolha:** Lê a entrada do usuário e a converte para maiúsculas para facilitar a comparação.
- **Switch case:** Utiliza um `switch case` para avaliar as opções válidas ("C" para continuar e "N" para novo jogo).
- **Retorno:** Retorna `true` se o jogador escolher continuar e `false` se escolher iniciar um novo jogo.
- **Tratamento de entrada inválida:** Imprime uma mensagem de erro caso o usuário digite uma opção inválida e repete a pergunta.



```
1  /**
2   * Le um numero int usado para definir se o jogador
3   * quer continuar jogando ou se deseja sair para
4   * o menu
5   *
6   * @return falso ou verdadeiro, a depender do
7   *         valor escolhido pelo usuario
8   */
9  public static boolean continuarJogo() {
10
11     while (true) {
12         System.out.print("Deseja continuar o jogo ou comecar um novo? (C - Continuar ou N - Novo Jogo): ");
13
14         String escolha = scan.nextLine().toUpperCase();
15
16         switch (escolha) {
17             case "C":
18                 return true;
19             case "N":
20                 return false;
21             default:
22                 System.out.print("Opcão invalida! Pressione ENTER...");
23                 limparBuffer();
24         }
25     }
26 }
```

Código do método continuarPartida().

limparConsole()

Limpa a tela do console, removendo qualquer texto ou elemento visual anterior.

Objetivos:

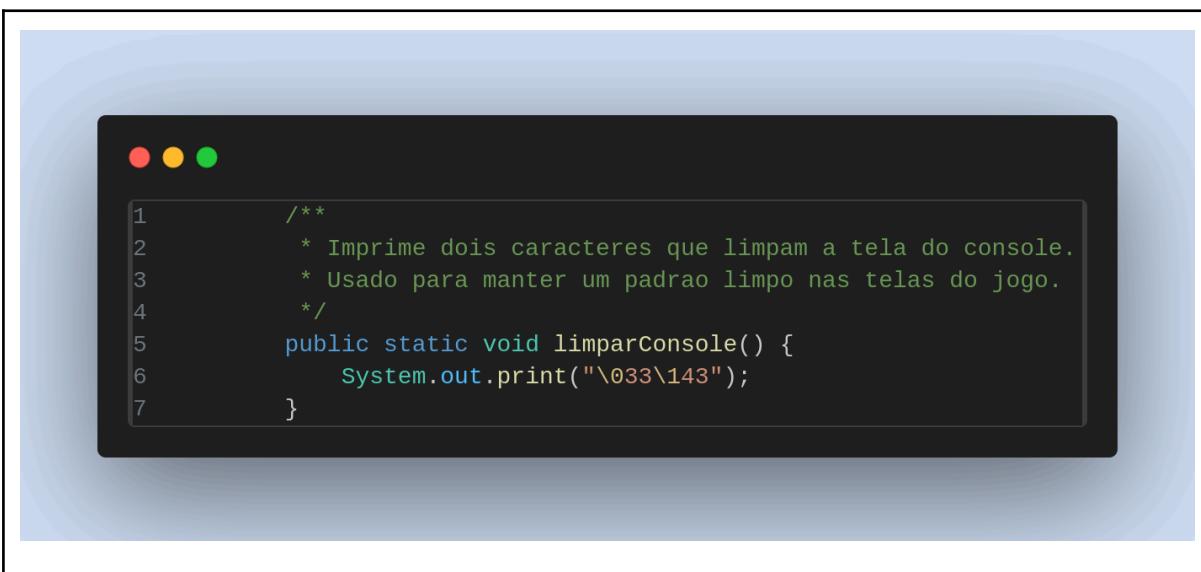
- Garantir uma interface de usuário **limpa e organizada**.
- Melhorar a experiência do jogador, evitando que **informações antigas interfiram na visualização do jogo**.

Detalhes da Implementação:

- **Sequência de escape ANSI:** Utiliza a sequência de escape ANSI "\033\143" para enviar um comando de limpeza de tela para o terminal.

Essa função não é essencial para o funcionamento do jogo, mas permite que o usuário tenha uma interface padronizada a cada execução (ao invés de ficar mexendo toda vez que uma jogada é feita).

Apesar de haver outras maneiras de implementar essa funcionalidade, essa foi a maneira que se encaixava melhor dentro do conhecimento da dupla de desenvolvimento.



```
1      /**
2       * Imprime dois caracteres que limpam a tela do console.
3       * Usado para manter um padrão limpo nas telas do jogo.
4       */
5      public static void limparConsole() {
6          System.out.print("\033\143");
7      }
```

Código do método limparConsole()

main()

Ponto de entrada principal do programa. Inicializa o jogo, cria o tabuleiro e apresenta o menu principal ao usuário.

Objetivos:

- **Inicialização:** Cria o tabuleiro do jogo.
- **Exibição:** Imprime o tabuleiro inicial para o usuário.
- **Menu principal:** Chama a função `menu()` para apresentar as opções disponíveis ao jogador.

Detalhes da Implementação:

- **Criação do tabuleiro:** Chama a função `criarTabuleiro()` para gerar a estrutura inicial do tabuleiro.
- **Exibição do tabuleiro:** Imprime o tabuleiro recém-criado na tela, utilizando a função `imprimirTabuleiro()`.
- **Mensagem de boas-vindas:** Chama a função `bemVindo()` para exibir uma mensagem de boas-vindas ao jogador.
- **Menu principal:** Chama a função `menu()` para apresentar as opções do jogo e iniciar a interação com o usuário.



```
1      /**
2       * Começa o programa, criando um tabuleiro de exemplo, imprimindo
3       * ele e chamando o menu() para o usuário
4       * @param args este parâmetro é um padrão do Java.
5       */
6      public static void main (String[] args) {
7
8          int[][] tabuleiro = criarTabuleiro();
9
10         System.out.println("Referência");
11         imprimirTabuleiro(tabuleiro);
12
13         bemVindo();
14         menu();
15     }
16 }
```

Código da função main().

Desenvolvimento

Abaixo é possível visualizar um fluxograma que demonstra como o jogo chama as funções. Devido a complexidade do código, o diagrama pode apresentar uma aparência um tanto confusa, para minimizar este problema, foram utilizadas cores diferentes para indicarem fluxos diferentes de execução.

Para visualizar com maior qualidade, é possível acessar o endereço no [github do projeto](#).

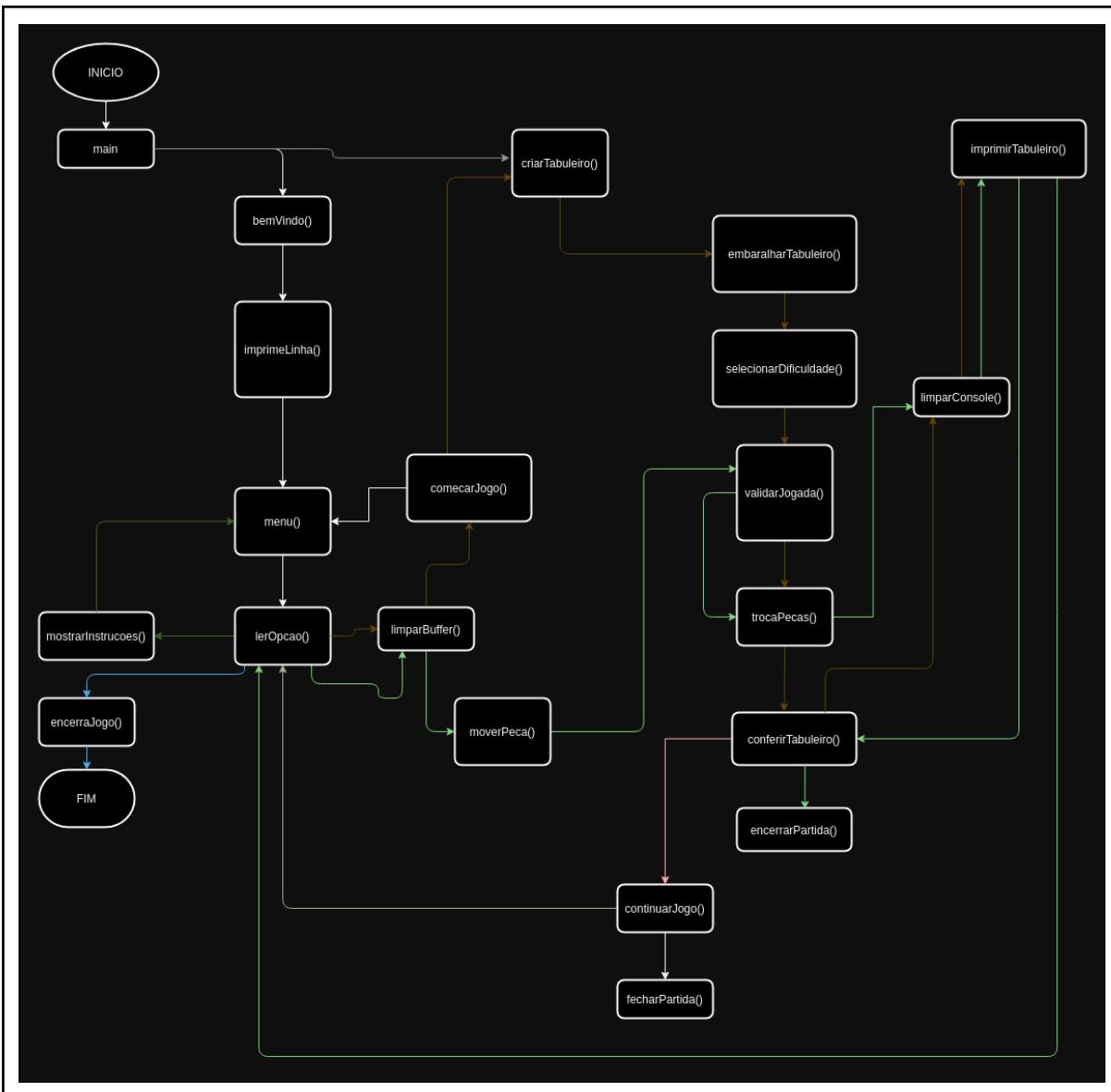


Diagrama do funcionamento do jogo SlidePuzzle desenvolvido para o trabalho.

Referências

- [Como Obter tamanho de Matriz? - Java - GUJ](#)
- [quebra-cabeça 3x3 - puzzle deslizante no SlidingTiles](#)
- [How to clear the console using Java? - Stack Overflow](#)
- [Collections \(Java Platform SE 6\)](#)
- [Gerador de números aleatórios em Java – como gerar números inteiros com Math Random](#)
- [Como fazer o programa esperar alguns segundos? : r/learnjava](#)
- [Iniciante em Java, Dúvida Operador Lógico de Negação ou NOT!](#)
- [Gostaria de entender como funciona o comando break. | Java JRE e JDK: compile e execute o seu programa | Alura](#)
- [Java Break - Javatpoint](#)
- [Múltiplos e divisores - Toda Matéria](#)
- [Como criar vetores em Java](#)
- [What is the difference between a rule switch and a regular switch in Java? - Stack Overflow](#)
- [Como funciona a classe Scanner do Java?,
<https://www.tabnews.com.br/joelcarneiro/por-que-e-necessario-fechar-o-scanner-em-java>](#)
- [Variáveis Globais e Locais em Java | Caffeine Algorithm](#)
- [Qual a vantagem de utilizar o static junto com scanner - Java - GUJ](#)
- [O que é: Comando de Encerramento • Napoleon](#)
- [Qual o comando em java para encerrar o aplicativo? \[RESOLVIDO\]](#)
- [Javadoc not working in windows command prompt << Solution Here >> - Codeforces](#)
- [Criação do Javadoc no VScode | Alura](#)
- [O que significa public static void main\(String\[\] args\)? - Stack Overflow em Português](#)
- [System.exit\(0\) e System.exit\(1 \) qual a diferença - Fórum DevMedia](#)
- [System.exit\(\) em Java](#)