

**INSTITUTO FEDERAL DE MINAS GERAIS – CAMPUS OURO BRANCO BACHARELADO EM SISTEMAS DE INFORMAÇÃO**

**MAYCON JESUS PINTO VITOR HUGO OLIVEIRA SANSON**

**TRABALHO PRÁTICO II:**

**POO E ESTRUTURAS DE ARMAZENAMENTO**

**Ouro Branco 2025**

|  |  |  |
| --- | --- | --- |
|  | **SUMÁRIO** |  |
| **1 INTRODUÇÃO** |  | **3** |
| **2 OBJETIVO** |  | **3** |
| **3 DESENVOLVIMENTO** |  | **3** |
| **4 DIFICULDADES ENCONTRADAS** |  | **7** |
| **5 SUGESTÕES FUTURAS** |  | **7** |
| **6 CONSIDERAÇÕES FINAIS** |  | **8** |
| **7 REFERÊNCIAS** |  | **9** |

1. **INTRODUÇÃO**

# O jogo de Tabuleiro:

O Banco Imobiliário é um dos jogos de tabuleiro mais populares do mundo, com sua origem remontando remontando a 1935, quando foi publicado nos Estados Unidos por Charles Darrow. Inspirado em "The Landlord's Game", criado por Elizabeth Magie, o jogo combina estratégia, negociação e administração financeira, proporcionando aos jogadores a experiência de comprar, vender e administrar propriedades em busca da vitória. Com diversas adaptações regionais, o Monopoly, seu nome original, continua a ser um ícone cultural, mantendo sua relevância ao longo das décadas.



Figura 1: Imagem extraída do site Wikipedia Brasil (https://upload.wikimedia.org/wikipedia/commons/7/78/Monopoly\_board\_on\_white\_bg.jpg)

# Objetivo

# Este trabalho tem como objetivo simular uma versão digital do jogo "Banco Imobiliário”. O trabalho permitiu a pratica de conceitos fundamentais de POO, como encapsulamento, herança e uso de estruturas de dados. A interação do jogo ocorrerá exclusivamente via terminal, permitindo a integração entre lógica de programação e aplicação dos conceitos aprendidos. A interação ocorre via terminal, consolidando conhecimentos de lógica de programação e reforçando o design modular para facilitar a reutilização e manutenção do código.

# Desenvolvimento

* 1. **Documentação e identificação do Trabalho**

## Funcionaento e fluxo de jogo

* + 1. Cadastro dos jogadores: Solicita-se ao usuário o número de participantes (de 2 a 6) e seus nomes.
    2. Movimentação: A cada rodada, os jogadores lançam dois dados, e o sistema calcula a soma para determinar o deslocamento no tabuleiro.
    3. Ações no tabuleiro: Dependendo da posição alcançada, o jogador pode comprar propriedades, pagar aluguéis, realizar melhorias ou enfrentar eventos aleatórios.
    4. Eliminação de jogadores: Caso um jogador não consiga quitar suas dívidas, ele é eliminado, e suas propriedades voltam a estar disponíveis.

## Entradas e Saídas

## Entradas:

* Quantidade inicial de jogadores e seus nomes.
* Comandos dos jogadores durante o jogo, como a decisão de comprar propriedades ou realizar melhorias.

## Saídas:

* Estado atualizado de cada jogador (saldo, propriedades, posição).
* Registro das rodadas e eventos do jogo.

# Execução e mecânica do jogo

1. **Ponto de entrada: Main.main():**

* A execução começa na classe Main, no método main.
* Este método chama App.startMenu(), que apresenta o menu inicial ao usuário.

1. **Menu Inicial (App.startMenu):**

O programa exibe um menu com as seguintes opções:

* **Adicionar Jogador**: Permite ao usuário registrar novos jogadores.
* **Iniciar Jogo**: Inicia o jogo com os jogadores cadastrados.
* **Sair**: Encerra o programa.

O usuário interage por meio de entradas no console (digitando números para selecionar opções).

1. **Cadastrando novos Jogadores:**

* Quando a opção "Adicionar Jogador" é escolhida:
* O programa solicita o nome do jogador.
* Cada jogador recebe um emoji único.
* Um novo objeto Player é criado e armazenado em uma lista no Game.

1. **Início do Jogo (Game.initGame)**

* O método makeBoard() é chamado para criar o tabuleiro:
* As 40 casas do tabuleiro são inicializadas, com tipos como Property, Luck, GoPrison, entre outros.
* Os jogadores são posicionados na primeira casa do tabuleiro (geralmente "Partida").
* O loop principal começa: enquanto houver mais de um jogador com saldo positivo, o jogo continua.

1. **Jogadas dos Jogadores**

* Cada jogador joga os dados (Dice) e se move no tabuleiro (Game.walkPlayer).
* Ao parar ou passar por uma casa, o comportamento associado àquela casa é executado. Por exemplo:
* **Propriedades (Property)**: O jogador pode comprar a propriedade ou pagar aluguel se outro jogador for o dono.
* **Sorte/Revés (Luck)**: Eventos aleatórios podem beneficiar ou prejudicar o jogador.
* **Imposto de Renda (IncomeTax)**: O jogador paga um valor fixo de imposto.
* **Lucros/Dividendos (Profit)**: O jogador ganha dinheiro.

1. **Mensagens e Atualizações**

* Durante cada jogada, mensagens relevantes (como pagamentos ou ganhos) são adicionadas à lista de mensagens (Messages).
* Ao final da jogada, as mensagens são exibidas e o tabuleiro atualizado.

1. **Fim do Jogo**
   * + Quando apenas um jogador permanece com saldo positivo, ele é declarado vencedor.

# Principais Classes e suas atribuições

# Package GAME:

**1. Classe Bank:**

* Atribuição: Gerencia transações financeiras, como aumentar ou diminuir o saldo de um jogador
* Métodos principais:
  + decreasePlayerBalance (Player player, int value): Diminui o saldo do jogador.
  + increasePlayerBalance (Player player, int value): Aumenta o saldo do jogador.

**2. Classe Dice:**

Atribuição: Simula dois dados para movimentação no tabuleiro.

Atributos:

number1 e number2: Valores dos dois dados.

Métodos principais:

getDice1(), getDice2(): Retornam os valores individuais.

getDicesSum(): Retorna a soma dos dois valores.

**3. Classe Game**

Atribuição: Gerencia a lógica central do jogo.

Métodos principais:

makeBoard(): Inicializa o tabuleiro com 40 casas.

walkPlayer(Player player, Dice dice): Move o jogador no tabuleiro e executa a lógica da casa onde ele parou.

initGame(): Prepara o jogo para começar.

newPlayer(String name): Adiciona um novo jogador.

Composição: Utiliza outras classes como Player, Space, e Dice.

**4. Classe Messages**

Atribuição: Armazena e exibe mensagens do jogo.

Métodos principais:

addMessage(String message): Adiciona uma mensagem à lista.

showAllMessagesStacked(): Exibe e limpa todas as mensagens acumuladas.

**5. Classe Player**

Atribuição: Representa um jogador no jogo.

Atributos:

name: Nome do jogador.

emoji: Identificador visual.

balance: Saldo inicial de $1500.

Métodos principais:

decreaseBalance(int value): Reduz o saldo.

increaseBalance(int value): Aumenta o saldo.

# Package Spaces

1. Space (Superclasse)

Função: Representa uma casa genérica no tabuleiro.

Métodos principais:

onPlayerStop(Player player, Dice dice): Define o comportamento ao parar na casa.

onPlayerGoesOver(Player player, Dice dice): Define o comportamento ao passar pela casa.

2. Subclasses de Space

Property:

Representa propriedades que podem ser compradas e alugadas.

Métodos: buyProperty, getRentValue.

GoPrison:

Envia o jogador à cadeia.

Métodos: Sobrescreve onPlayerStop para mover o jogador à prisão.

Luck:

Gera eventos aleatórios (sorte ou revés).

IncomeTax:

Cobra imposto de um jogador.

Profit:

Dá dinheiro ao jogador.

# Package UI

1. App

Função: Gerencia a interface do usuário.

Métodos principais:

startMenu(): Exibe o menu inicial.

menu(): Apresenta as opções do jogo.

printBoard(): Exibe o estado atual do tabuleiro.

2. Main

Função: Ponto de entrada do programa. Chama App.startMenu() para iniciar o jogo.

# public static void printTable()

Este método, ilustrado na Figura 2, é usado para exibir para o usuário a Tabela de jogo. Ele utiliza uma enumeração a *LinePrintMode,* no intuito de formatar as bordas para formatar as bordas e linhas do tabuleiro com caracteres da tabela ASCII, para a visualização atual estado da tela do jogo.

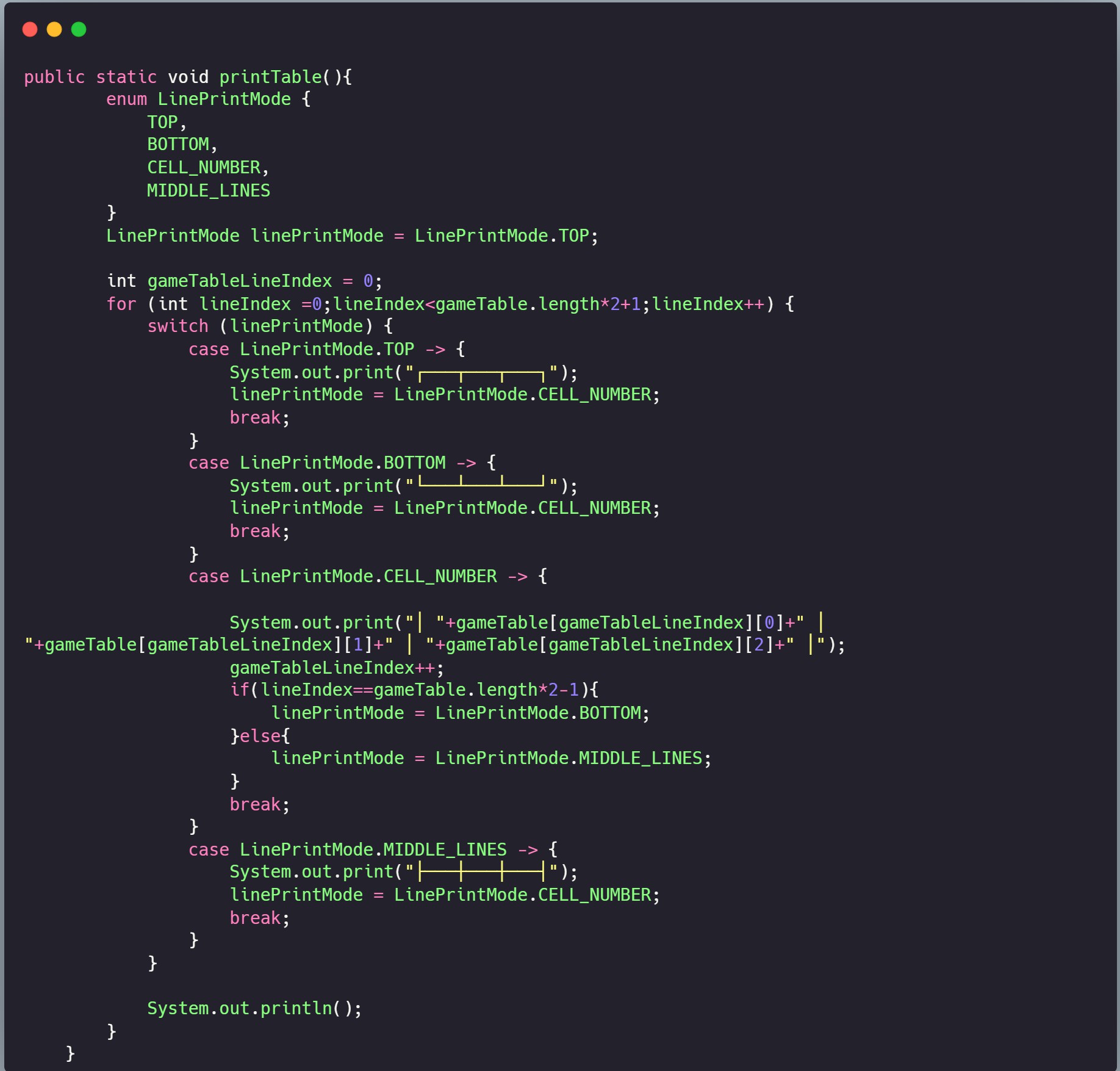


Figura 2: método printTable do programa Racha-Cuca.

# public static void move(PossibleMove target)

A Figura 3, ilustra, um método que é capaz de realizar um movimento: A troca da posição do espaço vazio com peça escolhida. Ele atualiza também a variável, *blankPosition*, que deverá conter a nova posição recebida pelo espaço vazio.

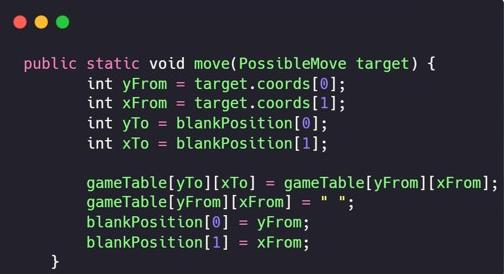


Figura 3: método move do programa Racha-Cuca.

# public static void shuffleTable(int difficulty)

Através do método ilustrado na Figura 4, é possível embaralhar a tabela de jogo no início, tornando o jogo mais desafiador. A quantidade de movimentos aleatórios depende do nível de dificuldade escolhido pelo jogador (fácil, médio ou difícil).

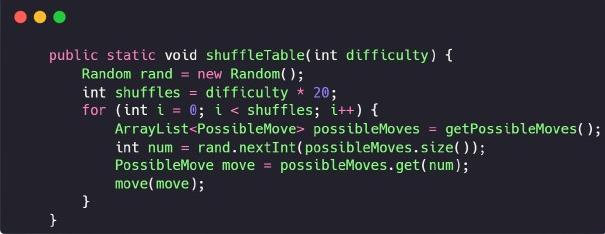


Figura 4: método shuffeTable do programa Racha-Cuca.

# public static boolean isGameSolved()

O método isGameSolved(), ilustrado pela Figura 5, verificará se o quebra-cabeças está resolvido. Faz isso, comparando a *gameTable()* com a configuração vencedora (*winningTable*). Retorna *true* (verdadeiro), caso a ordem esteja correta, caso contrário o retorno será *false* (falso).

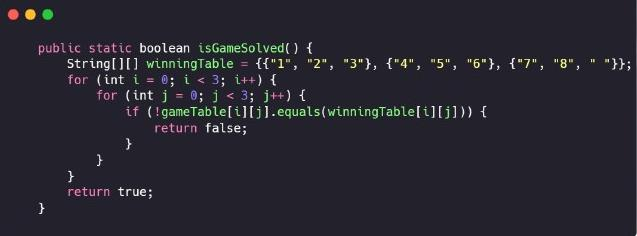


Figura 5: método move do programa Racha-Cuca.

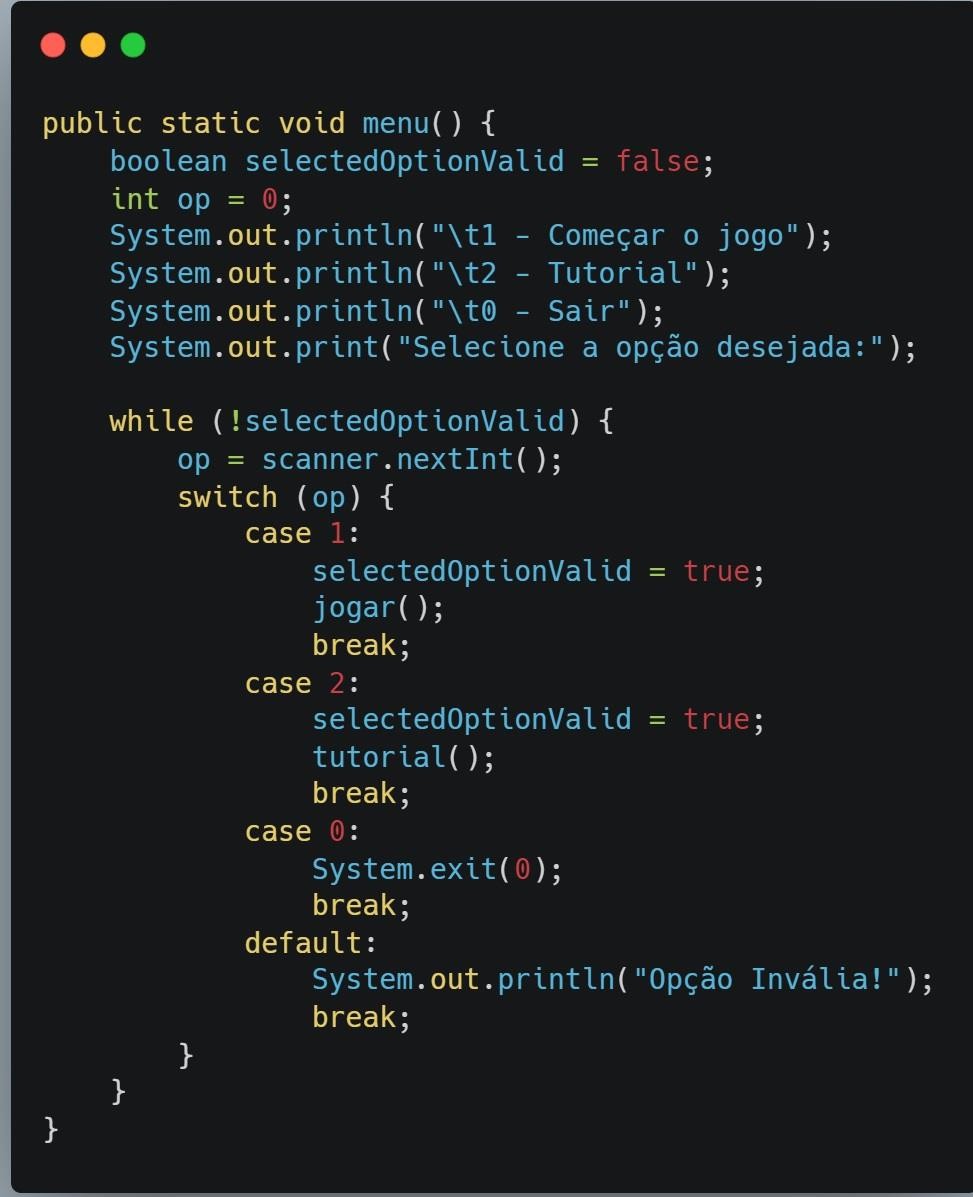
# void playerMove()

O método ilustrado pela Figura 6, é responsável por capturar o movimento feito pelo jogador. Ele pede que o jogador escolha uma peça para mover ao espaço vazio e também valida sua a escolha. Caso seja executado um movimento inválido o movimento for válido, o método executa o movimento e incrementa em uma unidade a variável *moveCouter*.



Figura 6: método move do programa Racha-Cuca.

* + 1. **public static void tutorial()**
    2. **public static void menu()**

****

* + 1. **public static void jogar()**

O método mostrado na Figura 7, é responsável por gerenciar o fluxo do jogo. Ele embaralha o tabuleiro com o nível de dificuldade escolhido, imprime o estado do jogo, e chama playerMove() repetidamente até que o jogo seja resolvido.

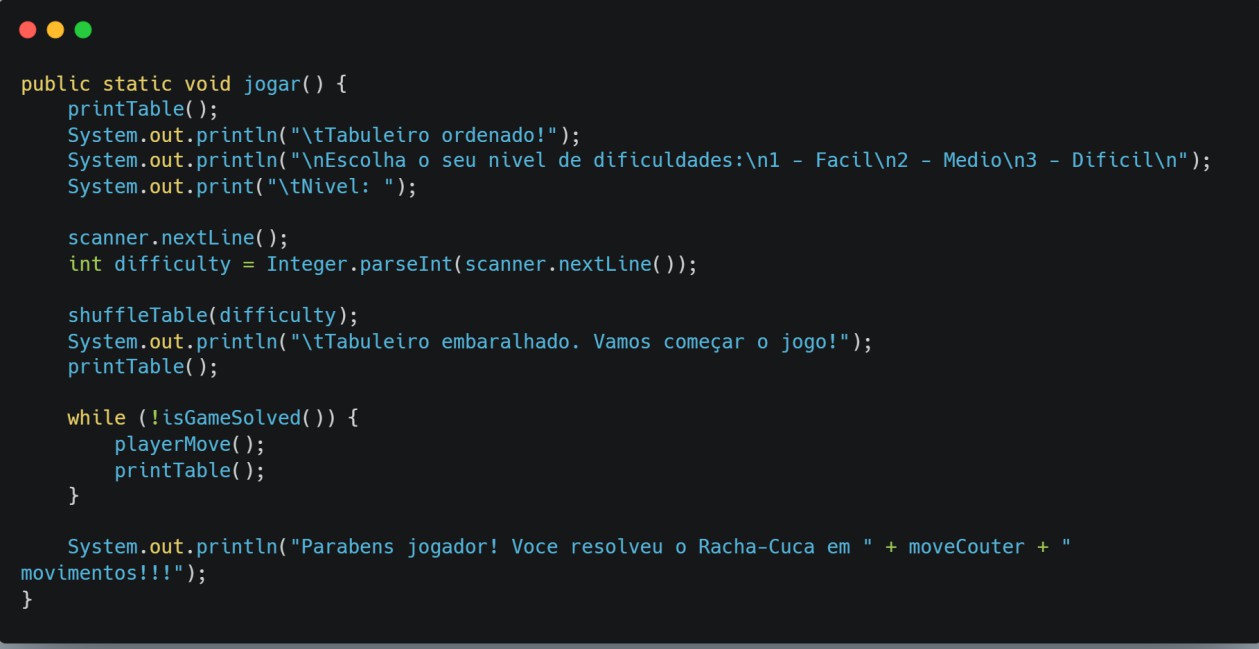


Figura 7: método jogar() do programa Racha-Cuca.

# public static void main(String[] args)

A Figura 8 ilustra o método principal inicializa o jogo, imprime o tabuleiro ordenado, embaralha o tabuleiro com base no nível de dificuldade, e mantém o loop até que o jogo seja resolvido. Exibe uma mensagem de vitória com o total de movimentos feitos pelo jogador.



Figura 8: método move do programa Racha-Cuca.

# Método getPossibleMoves()

O método getPossibleMoves(), retornará uma lista de movimentos possíveis a partir da posição atual contida no espaço vazio. Ele verifica as direções possíveis (cima, baixo, esquerda, direita) e adiciona à lista as posições válidas.

# public static ArrayList<PossibleMove> getPossibleMoves()

O método public static ArrayList<PossibleMove> getPossibleMoves() retorna uma lista de movimentos possíveis a partir da posição atual do espaço vazio na matriz 3x3 do jogo, ele é eficiente porque calcula todas as jogadas válidas de forma compacta, usando direções fixas (cima, baixo, esquerda, direita) e verificando rapidamente os limites da Matriz. Ele permite fácil acesso aos movimentos disponíveis, facilitando a lógica do jogo.

# Simulação do jogo

* + 1. **Menu Principal**

A Figura 9 ilustra a tela do **Menu Principal** Esta tela é o **Menu Principal** do jogo Banco IFMG, onde o jogador pode escolher entre três opções:

* + - * **0 - Sair**: Essa opção encerra o Menu do jogo.
      * **1 – Adicionar jogado:** Adiciona os jogadores que vão a partida.
      * **2 - Iniciar o jogo**: Após adicionar os jogadores adiciona inicia o jogo.

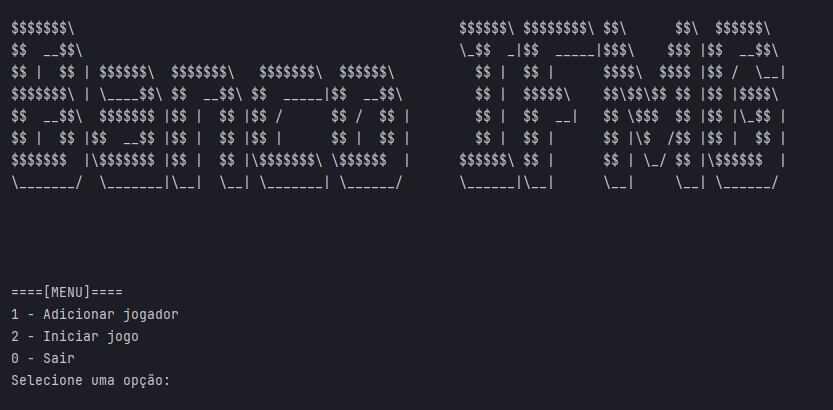


Figura 9: Tela inicial do Programa

# 3.2.1 Adicionar jogador

A partir da Figura 9 do **Menu Principal**, podemos acessar a opção de número 1 que abre a opção de **tutorial.** Esta tela é ilustrada pela figura 10. Nela o jogador inexperiente recebe dicas de como jogar.

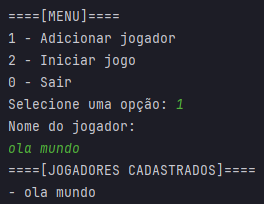


Figura 10: Tela de adiciona jogador

# 3.2.1 Menu Começar Jogo

As Figuras 11, 12 e 13 ilustram de forma mais aprofundada o funcionamento do jogo, após escolhermos a opção 2, o jogo começar:

O Tabuleiro do Jogo é exibido na tela em formato de grade, onde o jogador enxerga sua posição através de emojis coloridos.

A cada rodada o programa solicita ao jogador da vez que pressione enter para jogar os dados e avançar no tabuleiro.

O último jogador com saldo positivo no tabuleiro ganha.

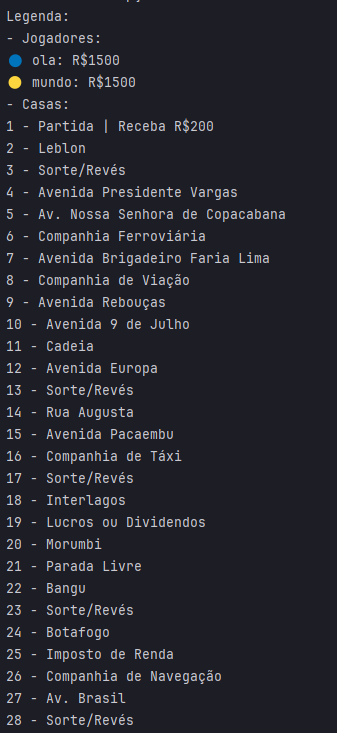


Figura 11: Tela de execução do jogo.

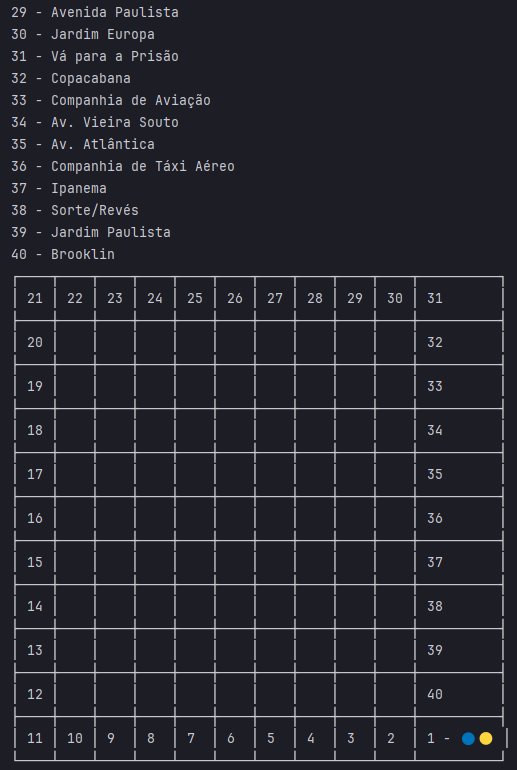


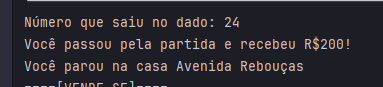
Figura 12: Tela de execução do jogo.



Figura 13: Tela de execução do jogo.

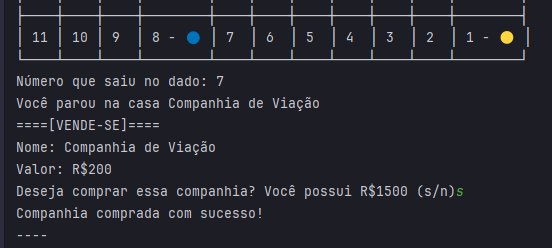
**Bônus ao passar pela partida**

Ao passar pela partida o jogador ganha R$200



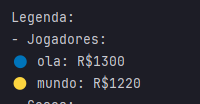
**Compra de propriedades**

Quando você para em uma propriedade sem dono é possível escolher se deseja comprá-la ou não, como demonstra a figura abaixo.



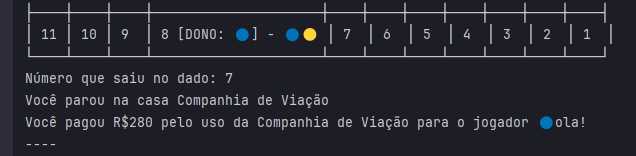
Após a compra o tabuleiro indica quem é o dono pelo emoji de circulo azul (que é apresentado na legenda a cada round) como no exemplo abaixo.





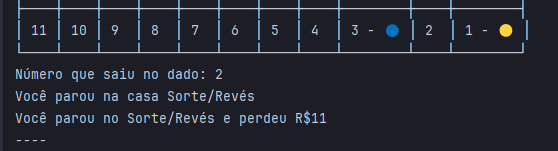
**Pagamento de aluguel ou pelo serviço**

Ao parar em uma casa que ja possui dono e que não é o jogador da rodada é pago um aluguel de acordo com o nivel da casa ou em caso de parada em companhias é pago de acordo com os valores do dado.



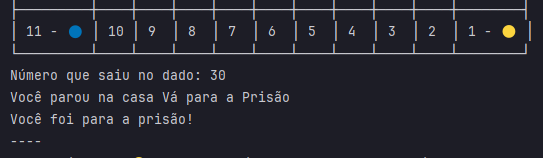
**Sorte/revés**

Ao parar nesta casa você pode ganhar de 1 até 150 reais ou perder de 1 a 80 reais



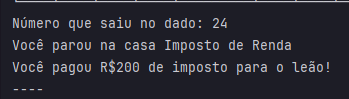
**Prisão**

Ao parar na casa vá para a prisão o jogador vai para em casa de prisão sem direito ao bónus monetário quando passa pelo inicio.



**Imposto de renda**

Ao parar na casa imposto de renda você paga R$200 de imposto de renda.



# Dificuldades encontradas

A maior dificuldade encontrada foi em relação ao tempo para o desenvolvimento do algoritmo, por não se tratar de um projeto trivial. Para concluir o projeto, foi necessário um tempo maior do que o demandado pelos algoritmos implementados até o momento.

1. **CONSIDERAÇÕES FINAIS**

A implementação dos códigos do “Banco Imobiliário” digital proporcionou uma experiência enriquecedora para o treino e aplicação dos conceitos da Orientação a Objetos em Linguagem Java. Dentre os principais desafios encontrados, destacam-se a validação de entradas de usuários e a lógica para calcular eventos aleatórios e aluguéis.

Embora o código seja funcional e ofereça uma base sólida, sempre há espaço para melhorias, especialmente na criação de uma interface gráfica para amigável para o usuário e reestruturação do código, que poderia usar mais classes para facilitar a resoluão de problemas. Um passo natural a partir deste segundo Trabalho Prático é continuar a evolução nos conceitos da disciplina, aprofundando o aprendizado e aprimorando as habilidades na Programação Orientada a Objetos.

1. **REFERÊNCIAS**

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da programação de computadores: algoritmos, Pascal e C/C++ (Padrão Ansi) e Java**. 3. ed. São Paulo: Pearson Education do Brasil, 2012. x, 569 p.

DEITEL, H. M; DEITEL, P. J. **Java: como programar**. São Paulo:Prentice Hall, 2010.

**CARBON** (Create and share beautiful images of your source code). Disponível em:

<https://carbon.now.sh/>. Acesso em: 29 de dezembro de 2024.

**ASCII table (ASCII Codes).** Disponível em: <<https://theasciicode.com.ar/>>. Acesso em 19 de dezembro de 2024.