João Nunes 47220
Miguel Marques 47204
Filipe Ribeiro 47195

3.º Trabalho Prático
05.01.2020

## Introdução

Neste trabalho realiza-se o jogo de tabuleiro conhecido como *Reversi*. O enunciado fornecido deu-nos uma base para começarmos a trabalhar. Primeiro, a implementação de uma consola customizada (ConsolePG.jar) facilita-nos o desenvolvimento do jogo. Segundo, é-nos fornecido também duas classes: *Reversi.java* que tem toda a informação do jogo, regras e execução de comandos (lógica do jogo) para que o jogo funcione corretamente; *Panel.java* é a outra classe dada, tem como objetivo implementar os aspetos mais estéticos do jogo e o que aparece no ecrã (apresentação do jogo).

Com isto a programação adicional foi só realizada na classe *Reversi.java* seguindo a ordem de tópicos proposta no enunciado.

## Análise de métodos e estruturas básicas

### Inicialização de variáveis globais

```
public static final int BOARD_DIM = 8, BOARD_TOTAL = BOARD_DIM * BOARD_DIM;
private static boolean terminate = false;

private static int cursorLine, indexLine, indexCol;
private static char cursorCol;
private static boolean player = true;
private static byte empty = 0, playerA = 1, playerB = 2, possibleplayerA = 3,
possibleplayerB = 4;

private static int lineLimit, colLimit;
private static int totalA = 2, totalB = 2;

private static byte[][] boardState = new byte[BOARD_DIM][BOARD_DIM];
```

`BOARD_DIM` é a dimensão do tabuleiro; `BOARD_TOTAL` é o número total de casas que o tabuleiro tem. A variável `terminate` define se o jogo acaba ou não (consola/janela fecha). `player` com o valor **`true`** indica o jogador A, em **`false`** indica o jogador B. `totalA` e `totalB` indica a pontuação de cada jogador (jogador A e jogador B respetivamente). `cursorLine` e `cursorCol` refere-se à posição atual em coordenadas do tabuleiro (“1..9” e “A..H”) em que o cursor se encontra (linha e coluna do cursor respetivamente). Para podermos processar a lógica do estado atual do jogo criámos um *array* bidimensional, `boardState`. Os índices para este *array* são `indexLine` e `indexCol` (índice da linha e índice da coluna respetivamente). Estas coordenadas em índice dão-nos a possibilidade de escrever e ler no *array* as informações seguintes:

- se a posição atual está vazia, `empty = 0`;
- se a posição atual tem uma peça do jogador A, `playerA = 1`;
- se a posição atual tem uma peça do jogador B, `playerB = 2`;
- se a posição atual seja uma possível jogada para o jogador A, `possibleplayerA = 3`;
- se a posição atual seja uma possível jogada para o jogador B, `possibleplayerB = 4`.

Por fim temos `lineLimit` e `colLimit` que nos ajudará mais tarde a desenvolver dois métodos – *searchArray()* e *flipPieces()*.

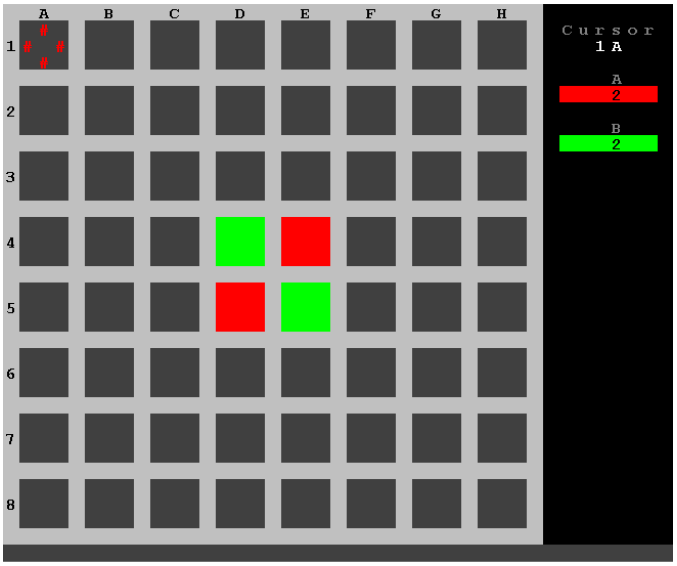
### Método *main()*

O método *main()* foca-se puramente em inicializar a consola, *Panel.init()*, imprimir mensagem "Welcome" e inicializar o método *playGame()*. Mais tarde, caso `terminate` mudar para **`true`** (o utilizador quer fechar a consola) é impresso outra mensagem "BYE" e a consola fecha-se, *Panel.end()*.

```
public static void main(String[] args)
    {
        Panel.init();
        Panel.printMessage("Welcome");
        playGame();
        Panel.printMessageAndWait("BYE");
        Panel.end();
    }
```

### Método *playGame()*

Este método, resumidamente, serve para inicializar o tabuleiro com as peças iniciais, colocar o cursor na posição (1,A) e esperar o próximo comando do utilizador (esperar que seja pressionada uma tecla).




```
int key;
int middle = BOARD_DIM / 2;
Panel.putPiece(middle, (char) ('A' + middle), true);
updateBoard(indexLine(middle), indexCol((char) ('A' + middle)), playerA);
Panel.putPiece(middle, (char) ('A' + middle - 1), false);
updateBoard(indexLine(middle), indexCol((char) ('A' + middle - 1)), playerB);
Panel.putPiece(middle + 1, (char) ('A' + middle), false);
updateBoard(indexLine(middle + 1), indexCol((char) ('A' + middle)), playerB);
Panel.putPiece(middle + 1, (char) ('A' + middle - 1), true);
updateBoard(indexLine(middle + 1), indexCol((char) ('A' + middle - 1)), playerA);
updateCursor(1, 'A');
```

Primeiro calcula-se o meio do tabuleiro, `middle`. De seguida começa-se a "desenhar" as peças no tabuleiro e a adicionar essa informação ao *array* `boardState`. Por exemplo, a primeira peça é colocada na coordenada (4, E) usando o método *Panel.putPiece()* e como coordenadas de índice (3,4) é também atualizado o `boardState[][]` usando o método *updateBoard()*. No fim o cursor é posto na coordenada (1,A) a partir do método *updateCursor()*.

```
do
{
    key = Console.waitKeyPressed(3000);

    if (key > 0)
    {
        processKey(key);
        Console.waitKeyReleased(key);
    } else Panel.clearMessage();
} while (!terminate);
```

Nesta parte espera-se pela ação do utilizador, ou seja, o programa"está à espera" que o utilizador pressione uma tecla. Para isso, utiliza-se um *do-while* com o objetivo de estar sempre à espera de um *input*. *Console.waitKeyPressed()* é o método que fica à espera até o utilizador premir uma tecla. De seguida, verifica-se se a tecla é válida (`key > 0`) e "processa-se" a tecla pressionada, *processKey()*. Se nada for pressionado em 3 segundos (`3000`) o programa apaga a última mensagem na consola na parte de baixo. Caso o jogador decidir sair do jogo (neste caso pressionando a tecla *ESC*) a condição `!terminate` fica falsa e "quebra" o *do-while loop* acabando assim o método *playGame()* e por sua vez realizando as últimas linhas em *main()* e fechando a consola no fim.

### Método *updateCursor()*

Este método atualiza a posição do cursor. Primeiro verifica se o cursor não passa dos limites do tabuleiro e de seguida muda o cursor no próprio tabuleiro (cursor composto por "#") usando o método *Panel.moveCursor()*.

```
private static void updateCursor(int line, int col)
{
    if (line<1 || line>BOARD_DIM || col<'A' || col>='A'+BOARD_DIM) return;
    cursorLine = line;
    cursorCol = (char) col;
    Panel.moveCursor(cursorLine,cursorCol,player);
}
```

### Método *processKey()*

Neste método usa-se um *switch-case* para escolher qual operação se realiza com base na tecla pressionada.

```
private static void processKey(int key)
{
    switch (key) {
        case VK_ESCAPE: terminate = Panel.confirm("Terminate game"); break;
        case VK_UP:     updateCursor(cursorLine-1,cursorCol); break;
        case VK_DOWN:   updateCursor(cursorLine+1,cursorCol); break;
        case VK_LEFT:   updateCursor(cursorLine,cursorCol-1); break;
        case VK_RIGHT:  updateCursor(cursorLine,cursorCol+1); break;
        case VK_SPACE:
        case VK_ENTER:  play(); break;
        default:
            if (key>='A' && key<'A'+BOARD_DIM) updateCursor(cursorLine,key);
            else if (key>='1' && key<'1'+BOARD_DIM) updateCursor(key-'0',cursorCol);
    }
}
```

Os casos possíveis são:

- `case VK_ESCAPE`: tecla *ESC*, termina o jogo (`terminate = true`);
- `case VK_UP`, `VK_DOWN`, `VK_LEFT`, `VK_RIGHT`: teclas cima, baixo, esquerda e direita, movem o cursor nas respetivas direções;
- `case VK_SPACE` e `VK_ENTER`: fazem ambos a mesma coisa, iniciar o método *play()*;
- `default`: este caso serve para o utilizador inserir as coordenadas manualmente.

**Método *play()***

```
private static void play()
{
    indexLine = indexLine(cursorLine);
    indexCol = indexCol(cursorCol);

    if (validatePlay())
    {
        scoreCount();
        Panel.printTotal(true, totalA);
        Panel.printTotal(false, totalB);
        player = !player;
        gameOverCheck();
    }
}
```

No início deste método atualizamos as variáveis globais, `indexLine` e `indexCol`, a partir do método *indexLine()* e *indexCol()* respetivamente. Deste modo, a partir do momento em que o utilizador pressiona *ENTER*, o cursor não se vai mexer mais até ao final da validação da jogada, dando oportunidade a fixar variáveis importantes. De seguida, num *if-statement*, faz-se a verificação da jogada na coordenada atual a partir do método *validatePlay()*. Se a jogada for válida, faz-se a contagem de pontos, *scoreCount()*, imprime-se na consola as pontuações de cada jogador, *Panel.printTotal()*, muda-se o valor da variável `player` para o outro jogador e de seguida testa-se se o jogo acabou ou não, *gameOverCheck()*.

**Método *indexLine()* e *indexCol()***

```
private static int indexLine (int line) { return line-1; }

private static int indexCol (char col) { return col-'A'; }
```

Estes dois métodos convertem as coordenadas do tabuleiro (“1..9”,”A..H”) para coordenadas índice para que se possa trabalhar com o *array* `boardState`.

Para as linhas é bastante fácil. Basta subtrair-mos uma unidade ao valor da coordenada no tabuleiro; assim seguimos as regras dos *arrays* em *Java*: a primeira casa de um *array* tem como índice 0. Para as colunas usamos o cálculo em *UNICODE*, sendo que para obtermos o valor em índice do *array* `boardState` teremos que subtrair por um valor constante que neste caso é a letra ‘A’. Esta letra serve como referência do alfabeto, por exemplo, se o valor decimal de ‘A’ for 68 e se a coordenada em que estamos for (1,C), se fizermos ‘C’-‘A’, o valor decimal que resulta é 2, sendo este a coordenada em índice que usaremos para `boardState[][]`.

**Método *boardLine()* e *boardCol()***

```
private static int boardLine (int line) { return line+1; }

private static char boardCol (int col) { return (char)(col+'A'); }
```

Este método faz o contrário dos métodos *indexLine()* e *indexCol()* respetivamente. Em vez de subtrair soma por 1 as linhas e por 'A' as colunas. Atenção que no caso das colunas é preciso converter para *char* para que possamos obter as coordenadas do tabuleiro corretamente.

**Método *currentPlayer()* e *enemyPlayer()***

```
private static byte currentPlayer() { return (player) ? playerA : playerB; }

private static byte enemyPlayer() { return (player) ? playerB : playerA; }
```

Este método serve unicamente para indicar qual jogador está a jogar de acordo com a informação no *array* `boardState`. Isto é: se a variável `player` for **`true`** significa que o jogador A está a jogar; assim *currentPlayer()* vai retornar o valor de `playerA`, que neste caso é 1. O mesmo acontece em *enemyPlayer()*. Se `player` estiver a **`true`** significa que o jogador inimigo na jogada atual é o jogador B sendo que o valor que este método retorna é o valor de `playerB`, que neste caso é 2.

**Método *updateBoard()***

```
private static void updateBoard(int line, int col, byte player)
{
      boardState[line][col] = player;
}
```

Este método serve para atualizar a informação no *array* `boardState` na posição atual (a posição em que se pressionou *ENTER*. O método recebe 3 variáveis: `line`, linha em que se encontra o cursor (já convertida para índice); `col`, coluna me que se encontra o cursor (já convertida para índice); `player` (`playerA` ou `playerB` de acordo com o jogador que esteja a jogar no momento).

## Análise de estruturas e métodos complexos

### Método *scoreCount()*

Este método conta a pontuação de cada jogador após ter havido alterações ao estado do tabuleiro.

```
private static void scoreCount ()
{
    totalA = 0;
    totalB = 0;

    for (int line = 0; line < BOARD_DIM; line++)
    {
        for (int col = 0; col < BOARD_DIM; col++)
        {
            if (boardState[line][col] == playerA)
                totalA++;
            else if (boardState[line][col] == playerB)
                totalB++;
        }
    }
}
```

São zeradas as duas variáveis globais, `totalA` e `totalB`. De seguida percorre-se o *array* `boardState` a partir de dois *for-loop*'s. Dentro desses dois *for*'s verifica-se que peças estão em cada posição e a que jogador pertencem. Se uma peça pertencer ao jogador A incrementa-se por uma unidade a variável `totalA`. Se uma peça pertencer ao jogador B incrementa-se por uma unidade a variável `totalB`. Existe a possibilidade de melhorar este método: em vez de percorrer o *array* inteiro cada vez que se vai contar a pontuação de cada jogador podia-se seguir cada jogada de modo a somar ou subtrair os pontos necessários.

### Método *validatePlay()*

```
if (boardState[indexLine][indexCol] != empty)
    posEmpty = false;
else
{
    for (int option = 0; option < 8; option++)
    {
        if (validSearch(lineDir(option), colDir(option)))
        {
            validPlay = true;
            flipPieces (lineDir(option), colDir(option));
        }
    }
}
```

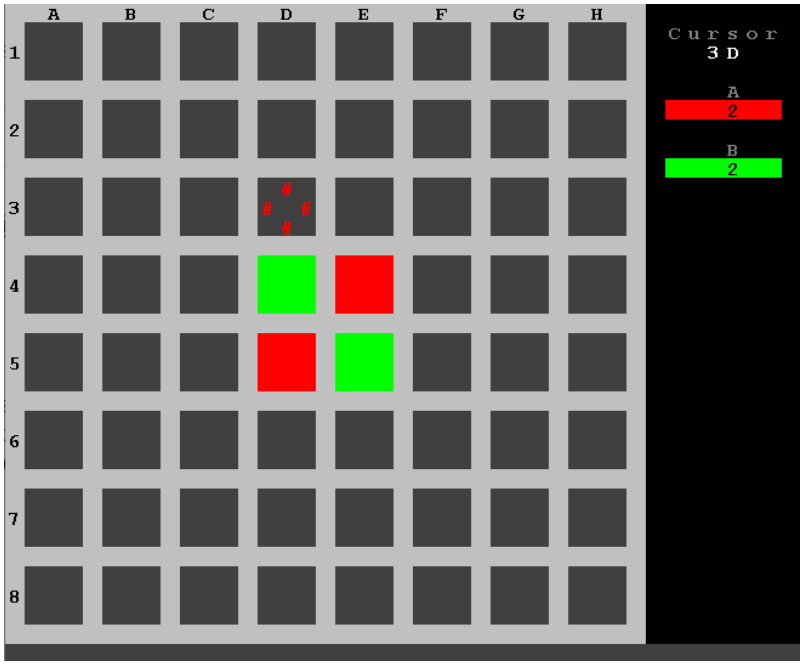
Antes de tudo, este método, para além de verificar se a jogada é possível, realiza-a caso seja. Isto é, primeiro, procura se a posição está vazia ou não. Se não estiver vazia altera `posEmpty` para **`false`** (`posEmpty` : posição vazia ou posição sem peça). Se a posição estiver vazia "entra" na outra hipótese do *if-else* acima. Este *if-else* contém um *for-loop* que percorre todas as direções possíveis a pesquisar (cima-esquerda, cima, cima-direita, direita, baixo-direita, baixo, baixo-esquerda, esquerda). Dentro deste *for-loop* existe outro *if-statement* que contém um método chamado *validSearch()*. Este método vai pesquisar nessas 8 direções e retornar se existe pelo menos UMA jogada possível (jogada em que se vire pelo menos UMA peça). Se a pesquisa for válida então `validPlay` torna-se **`true`** e viram-se as peças que são possíveis virar, *flipPieces()*.

Por fim, verifica-se se existe alguma jogada válida. Caso não exista, imprime-se "Invalid play" na consola e retorna-se o valor de `validPlay`, que neste caso é **`false`**, ao método no qual este foi chamado (*play()*). Caso exista, retorna-se simplesmente o valor de `validPlay`, que neste caso é **`true`**.

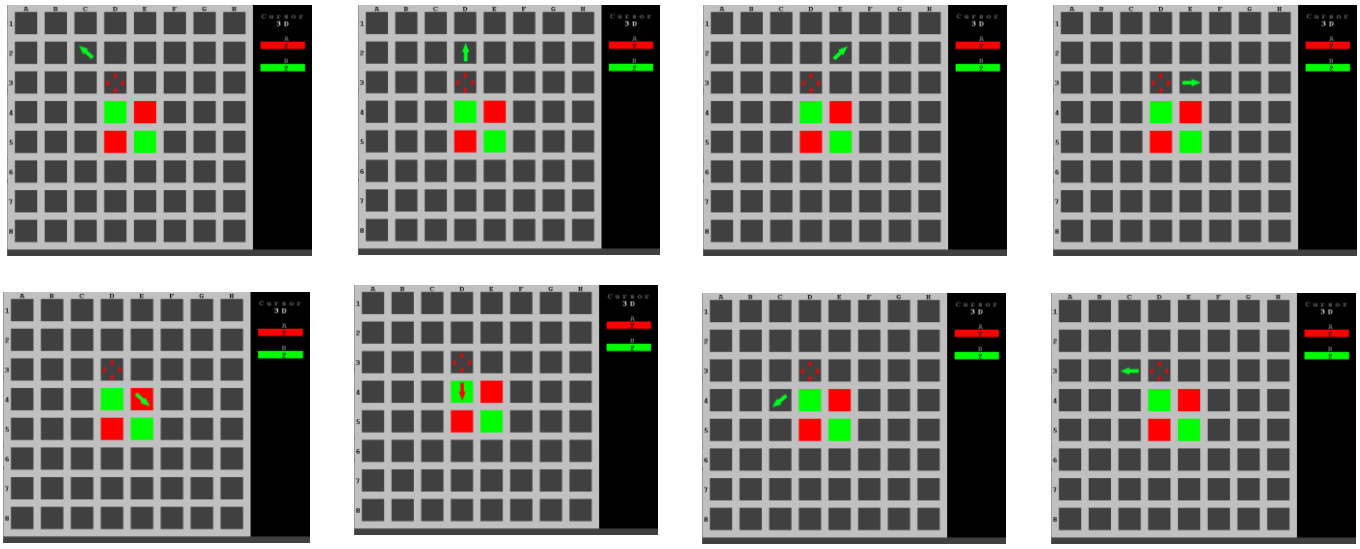
```
if (!posEmpty || !validPlay)
    Panel.printMessage("Invalid play");

return validPlay;
```

### Método *lineDir()* e *colDir()*

Para continuar a explicar o resto do código é necessário explicar a forma como pesquisamos no nosso tabuleiro usando duas direções: direção da linha (`lineDir`) e direção da coluna (`colDir`). Imaginemos a seguinte situação:




O cursor encontra-se na coordenada (3,D). Em coordenadas de índice: (2,3). O utilizador pressiona *ENTER*. Para sabermos sequer se é uma jogada possível precisamos de pesquisar à volta do cursor. Implementamos um sistema de 8 direções a serem percorridas no *array* `boardState`:



Por ordem, as direções são: cima-esquerda, cima, cima-direita, direita, baixo-direita, baixo, baixo-esquerda, esquerda. Por outras palavras, em coordenadas de índice, as coordenadas são (por ordem): `lineDir`: -1,-1,-1,0,0,1,1,1; `colDir`: -1,0,1,-1,1,-1,0,1. Estes números são os que se somam à coordenada atual, funcionam como um incremento de um *for-loop*.

Com isto explicado, estes dois métodos percorrem todas essas direções de acordo com uma variável `option` já referida anteriormente. Assim este método recebe a variável `option` do método *validatePlay()* e assim escolhe qual a direção em causa. Se `option == 2` significa que estamos a percorrer na direção cima-direita (consultar código-fonte e esquema acima).

```
private static int lineDir (int option)
{
    int lineDir;

    switch (option)
    {
        case 0:
        case 1:
        case 2:
            lineDir = -1;
            return lineDir;
        case 3:
        case 4:
            lineDir = 0;
            return lineDir;
        case 5:
        case 6:
        case 7:
            lineDir = 1;
            return lineDir;
        default:
            return 0;

    }
}
```

```
private static int colDir (int option)
{
    int colDir;

    switch (option)
    {
        case 0:
        case 3:
        case 5:
            colDir = -1;
            return colDir;
        case 1:
        case 6:
            colDir = 0;
            return colDir;
        case 2:
        case 4:
        case 7:
            colDir = 1;
            return colDir;
        default:
            return 0;
    }
}
```

**Método *invertDir()***

Este método inverte a direção na qual se percorre. É constituído por um simples *switch-case* no qual retorna o valor inverso da direção. O **`default`** retorna 0 já que os únicos valores possíveis para as direções são -1, 0 e 1. Se retornar 0, no incremento da direção não irá afetar a posição atual.

```
private static int invertDir (int
countDir)
{
    switch(countDir)
    {
        case -1:
            return 1;
        case 1:
            return -1;
        default:
            return 0;
    }
}
```

### Método *validSearch()*

Este método engloba dois métodos adicionais: *proximitySearch()* e *searchArray()*.

```
private static boolean validSearch (int lineDir, int colDir)
{
    boolean validSearch = false;

    if (proximitySearch (lineDir, colDir))
        if (searchArray (lineDir, colDir))
            validSearch = true;
    return validSearch;
}
```

Inicia com um variável local, `validSearch`. Esta variável é a que irá ser retornada no fim do método. Segue-se um *if-statement* com o método `proximitySearch()` ("pesquisa das redondezas") e outro *if* com *searchArray()*. Se ambos forem **`true`**, verifica-se uma pesquisa válida (pesquisa em que existe pelo menos UMA jogada válida).

### Método *proximitySearch()*

Este método, mais uma vez, engloba dois outros métodos: *borderCheck()* e *searchPos()*.

```
private static boolean proximitySearch (int lineDir, int colDir)
{
    boolean validSearch = false;

    if (borderCheck (lineDir, colDir))
        if (searchPos(lineDir, colDir) == enemyPlayer())
            validSearch = true;

    return validSearch;
}
```

Semelhante ao *validSearch()*, este método consiste em eliminar direções que "não valem a pena procurar". Este método tem o objetivo de acelerar o processo de pesquisar no *array* `boardState`.

### Método *borderCheck()*

Este método retorna um booleano caso esteja na borda do tabuleiro (**`false`**) ou não (**`true`**).

```
private static boolean borderCheck(int lineDir, int colDir)
{
      return (indexLine + lineDir != -1 && indexLine + lineDir != BOARD_DIM
      && indexCol + colDir != -1 && indexCol + colDir != BOARD_DIM);
}
```

Para testar isso, adiciona à coordenada atual a direção a ser pesquisada (`lineDir` ou `colDir`). Se este novo valor exceder os limites do tabuleiro (`BOARD_DIM` e 0) então a condição torna-se imediatamente falsa. Se nenhum valor exceder os limites então a condição é verdadeira.

**Método *searchPos()***

```
private static byte searchPos (int lineDir, int colDir)
{
      return boardState[indexLine+lineDir][indexCol+colDir];
}
```

Este método retorna a primeira posição a ser pesquisada. Isto serve unicamente para, quando voltar ao método em que foi chamado, este comparar com o método *enemyPlayer()* num *if-statement* e verificar se é de facto uma posição com uma peça inimiga.

**Método *searchArray()***

Este é o método mais importante do código. Pesquisa, como diz o nome, na direção dada e a partir da posição atual, se existe pelo menos UMA peça a virar.

```
int pieceCounter = 0;
boolean validPlay = false;
```

No início deste método declaram-se as variáveis: `pieceCounter` e `validPlay`. `pieceCounter` serve para contar quantas peças se vão virar. `validPlay` serve como variável de retorno caso existe jogada válida ou não.

```
for (int lineToSearch = indexLine+lineDir, colToSearch = indexCol+colDir;
     (lineToSearch != -1 && lineToSearch != BOARD_DIM) && (colToSearch != -1
&& colToSearch != BOARD_DIM);
     lineToSearch += lineDir, colToSearch += colDir)
{
```

Após a declaração de variáveis, inicia-se um *for-loop*. Cria-se a variável `lineToSearch` (linha a pesquisar) e `colToSearch` (coluna a pesquisar); a condição é semelhante à do método *borderCheck()*, só que `lineToSearch` e `colToSearch` são variáveis dinâmicas (em constante alteração); por fim realiza-se a incrementação a partir da direção dada.

```
if (boardState[lineToSearch][colToSearch] == enemyPlayer())
    pieceCounter++;
else if (boardState[lineToSearch][colToSearch] == currentPlayer() &&
pieceCounter > 0)
{
    lineLimit = lineToSearch;
    colLimit = colToSearch;
    validPlay = true;
    break;
}
else
{
    validPlay = false;
    break;
}
```

Neste *if-statement* testa-se se a posição em questão (`lineToSearch`, `colToSearch`) tem uma peça inimiga ou não. Se tem, incrementa-se `pieceCounter`; se não tem e `pieceCounter = 0` então a jogada não é possível (validPlay = false). O outro caso é se a peça em questão é do jogador a jogar e `pieceCounter > 0`. Se assim for, atualizam-se as variáveis globais, `lineLimit` e `colLimit` e `validPlay` passa a **`true`**. Assim `lineLimit` e `colLimit` serve para sabermos a partir da posição atual até que coordenada é que temos de virar peças. Por fim este método retorna o valor final de `validPlay`.

### Método *flipPieces()*

Este método vira as peças. Começa por virar a peça no local do cursor. De seguida percorre da linha/coluna limite (`lineLimit`/`colLimit`) até à posição do cursor (isto na direção inversa). Ao percorrer essas posições vai virando as peças a partir do método *Panel.flipPiece()*. Dentro dor for-loop, temos em conta a direção inversa àquela que procurou no método searchArray(). Atualiza-se também o boardState[][] com o jogador a jogar, currentPlayer().

```
private static void flipPieces (int lineDir, int colDir)
{
    Panel.putPiece(cursorLine, cursorCol, player);
    updateBoard(indexLine, indexCol, currentPlayer());

    for (int lineToFlip = lineLimit + invertDir(lineDir), colToFlip =
colLimit + invertDir(colDir);
         (lineToFlip != indexLine) || (colToFlip != indexCol);
         lineToFlip += invertDir(lineDir), colToFlip += invertDir(colDir))
    {
        boardState[lineToFlip][colToFlip] = currentPlayer();
        Panel.flipPiece (boardLine(lineToFlip), boardCol(colToFlip));
    }
}
```

### Método *gameOverCheck()*

Neste método verifica-se se o jogo acaba ou não. Isto acontece se ambos os jogadores não conseguirem jogar ou o tabuleiro está cheio de peças.

Iniciam-se duas variáveis booleanas: `firstCheck` e `secondCheck`. Estas variáveis representam essas duas situações de fim de jogo.

```
boolean firstCheck = false,
secondCheck = false;
```

```
if (!possiblePlays())
{
    player = !player;
    if (!possiblePlays())
        firstCheck = true;
}

resetPossiblePlays();
```

`firstCheck` inclui o teste para cada jogador se tem jogadas válidas no tabuleiro. Testa para o *currentPlayer()*.Se este não tem jogadas possíveis troca para o outro. Se ambos não tiverem jogadas possíveis, `firstCheck` torna-se **`true`**. Para sabermos se existem jogadas possíveis usamos o método *possiblePlays()*. Como este método altera valores em `boardState[][]`, no fim do processo temos que voltar a por os valores a 0 (`empty`) para que se possa continuar o jogo sem quaisquer problemas. Isto é explicado melhor no método *possiblePlays()*.

O `secondCheck` verifica se o tabuleiro está cheio. Declara-se um contador de peças, `withPieceCounter`. Com dois *for-loop*'s percorre-se o *array* `boardState` com a condição: se não está vazio, incrementa-se `withPieceCounter`.

```
int withPieceCounter = 0;
for (int line = 0; line < BOARD_DIM; line++)
{
    for (int col = 0; col < BOARD_DIM; col++)
    {
        if (boardState[line][col] != empty)
            withPieceCounter++;
    }
}
if (withPieceCounter == BOARD_TOTAL)
    secondCheck = true;
```

No fim testa-se se o contador de peças é igual ao número total de peças do tabuleiro (`BOARD_TOTAL`). Se este caso for verdadeiro, então o `secondCheck` torna-se **`true`**.

```
if (firstCheck || secondCheck)
{
    Panel.printMessageAndWait("Game over");
    terminate = Panel.confirm("Terminate game");
```

Por último, com um simples *if-statement*, verifica-se se existe alguma situação de fim de jogo. Caso haja, imprime-se a mensagem "Game over" na consola e termina o jogo (`terminate = true`).

**Método *possiblePlays()***

Este método muda os espaços vazios (`empty`) para jogadas possíveis (`possibleplayerA`/`possibleplayerB`) do jogador selecionado (`player`).

```
private static boolean possiblePlays()
{
    boolean playerCanPlay = false;

    for (int line = 0; line < BOARD_DIM; line++)
    {
        for (int col = 0; col < BOARD_DIM; col++)
        {
            if (boardState[line][col] == empty)
            {
                indexLine = line;
                indexCol = col;
                for (int option = 0; option < 8; option++)
                {
                    if (validSearch(lineDir(option), colDir(option)))
                    {
                        boardState[line][col] = (player) ? possibleplayerA : possibleplayerB;
                        playerCanPlay = true;
                    }
                }
            }
        }
    }

    return playerCanPlay;
}
```

Para isso, inicia-se uma variável booleana que serve de retorno deste método, `playerCanPlay`. Esta variável indica se o jogador em causa pode jogar ou não. De seguida com dois *for-loop*'s percorre-se `boardState[][]` e verifica-se que posições estão vazias. As que estiverem vazias, são testadas se são possíveis jogadas do jogador. Esta parte usa o método *validSearch()* e é semelhante quando estávamos a validar a jogada selecionada pelo utilizador. Se *validSearch()* valida a jogada, muda a posição para `possibleplayerA`/`possibleplayerB` tendo em conta o valor de `player` (utilização de operação ternária).

Assim que haja pelo menos UMA jogada possível `playerCanPlay` muda para **`true`**. Temos de ter em conta que isto altera o *array* `boardState` com os valores `possibleplayerA` (que neste caso é 3) e `possibleplayerB` (que neste caso é 4) logo, para que o programa continue a ler de forma correta o *array*, é necessário voltar a "zerar" (`empty`) essas posições que possivelmente poderiam ser jogadas.

### Método *resetPossiblePlays()*

Tal como referido no texto anterior, é necessário voltar a "esvaziar" as posições que poderiam ser jogadas. Imaginemos que o tabuleiro está cheio de "peças fantasma" da cor correspondente do jogador (vermelho ou verde) com o único uso de ver se existem jogadas válidas. Mas para continuar teremos que removê-las. Este método faz um "reset" parcial ao tabuleiro, retirando essas "peças fantasma".

```
private static void resetPossiblePlays()
{
    for (int line = 0; line < BOARD_DIM; line++)
    {
        for (int col = 0; col < BOARD_DIM; col++)
        {
            if (boardState[line][col] == possibleplayerA ||
boardState[line][col] == possibleplayerB)
                boardState[line][col] = empty;
        }
    }
}
```

Mais uma vez, com dois *for-loop*'s percorremos o *array* `boardState` de forma a procurar as posições que tenham o valor de `possibleplayerA` ou `possibleplayerB` e, se tiverem, voltar a preencher com o valor de `empty`.

## Conclusão

Ao acabar os pontos obrigatórios deste trabalho chegámos à conclusão que o código implementado está bastante compacto e preparado para receber pontos adicionais que são referidos no enunciado. Outro aspeto a ter em conta é o facto de não termos criado uma nova classe (por exemplo, a classe *Board*, tal como se refere no enunciado). A justificação baseia-se no tamanho do código escrito e na lógica por trás do jogo. Para os pontos obrigatórios a complexidade do jogo não requer uma grande atenção em termos de escrita e leitura de código. Sentimos até que seja um grande exercício fazer o código-fonte o mais compacto possível e todo na mesma classe.

Com isto tudo, o jogo Reversi fica completo com todos os pontos obrigatórios propostos. Estes pontos foram seguidos pela ordem indicada tal como outras dicas referidas no enunciado (a não repetição de código, a não utilização de "números mágicos", etc…).

Se pontos opcionais forem adicionados será também adicionado a este relatório a sua explicação e desconstrução de cada método.

## Pontos Adicionais

Como pontos adicionais, decidimos implementar os seguintes:

- Marcar as posições onde é possível jogar
- Quando termina o jogo perguntar se quer jogar outro.
- Quando premia a tecla 'N' começa um novo jogo, após confirmação.

### Método *possiblePlayMarks()*

```
private static void possiblePlayMarks(boolean PutMark) //Put = true, Clear = false;
{
    possiblePlays();
    for (int line = 0; line < BOARD_DIM; line++)
    {
        for (int col = 0; col < BOARD_DIM; col++)
        {
            if (boardState[line][col] == (player ? possibleplayerA : possibleplayerB))
            {
                if (PutMark)
                    Panel.putMark (boardLine(line), boardCol(col), player);
                else
                    Panel.clearMark (boardLine(line), boardCol(col));
            }
        }
    }
    resetPossiblePlays();
}
```

Este método recebe uma variável, `PutMark`, que, caso seja **`true`** significa que está a pôr as marcas das jogadas possíveis no tabuleiro. Caso seja **`false`**, está a limpar as marcas das jogadas possíveis para o jogador correspondente.

### Método *newGame()*

```
while (newgame)
{
    playGame();
    newGame();
}
```

Para começarmos um novo jogo temos primeiro que retornar ao método *main()*.

Este **`while`** no método *main()* serve para podermos estar sempre a jogar novos jogos dependendo da variável `newgame`.

Esta variável é alterada no dito método *newGame()*

*newGame()* verifica se de facto é para fazer um novo jogo. Se sim, entra no método *resetBoard()*.

```
private static void newGame()
{
    newgame = Panel.confirm("Play Again");
    terminate = !newgame;
    if(newgame)
        resetBoard();
}
```

```
private static void resetBoard()
{
    player = true;
    for (int line = 0; line < BOARD_DIM; line++)
    {
        for (int col = 0; col < BOARD_DIM; col++)
            boardState[line][col] = empty;
    }
    possiblePlayMarks(false);
    Panel.printGrid();
}
```

Para que o jogo recomece de acordo com as regras iniciais impostas é necessário ter em conta também o outro ponto adicional: recomeçar o jogo premindo a tecla 'N'.

Como já foi dito, é necessário retornar ao método *main()*. Para isso acontecer é necessário `terminate = true`.

Esta variável só toma o valor true quando o jogo de facto termina sendo que quando pressionamos 'N' para recomeçar, o jogo não terminou totalmente.

```
case VK_N:      newGame(); terminate = newgame; break;
```

É por isso que, quando se pressiona a tecla N, `terminate = newgame` porque para haver um novo jogo, o jogo atual tem de terminar.