**2D Arrays Workshop**

The aim of this workshop is to practice 2D array processing.

We will achieve this goal by implementing and analyzing a series of exercises and functions designed to manipulate and understand 2D arrays effectively.

This includes hands-on coding tasks and theoretical discussions to build a deeper understanding of 2D array structures and operations.

The files:

* This docx workshop content file.
* The example and warmup file: “WS5Warmup.java” (which you can and should provide to the students [even by sending it in the zoom chat]).
* The solution file: “WS5WarmupSolutions.java”.
* The workshop file: “WS5.java” (which you can and should provide to the students [even by sending it in the zoom chat]).
* The solution file: “WS5Solution.java”.

**Introduction – 2D array definition and initialization:**

(file: “WS5Warmup.java”)

Go through an example of initialization of a 2D array.

Make sure all students are familiar.

**#Question:** what are the default values in 2D arrays?

**#Answer:** The same as in 1D arrays – because a 2D array is an array of 1D arrays.

**Printing 2D arrays**

(file: “WS5Warmup.java”)

**#Exercise:** Ask students to take ~4 minutes and implement the function “print”:

public static void print(int[][] arr)

The function should print the contents of the 2D array, separated by whitespaces.

**#Note**: it is not enough to write System.out.println(arr) and adding space between the elements)

**#Solution**: The solution should look like this:

public static void print(char[][] board) {

for (int i = 0; i < board.length; i++) {

for (int j = 0; j < board[i].length; j++) {

System.out.print(board[i][j] + " ");

}

System.out.println();

}

}

**Changing element in a 2D array**

(file: “WS5Warmup.java”)

**#Exercise**: Ask the students to implement the function “initializeChangePrint”:

public static void initializeChangePrint()

The function should initialize a new 2D array using the “initializeExample” function provided, then change the item in indices [1][1] to 1, and then print the new changed 2D array using the “print” function they implemented earlier.

**#Solution:**

public static void initializeChangePrint() {

int[][] arr = initializeExample();

arr[1][1] = 1;

print(arr);

}

**Introduction to chess:**

Start the workshop by explaining the setup and basics moves.

!(Notice that these are partial rules because these are the relevant rules for this workshop)!

**Setup**:

* The board is an 8x8 grid, with alternating white and black squares.
* Each player starts with 16 pieces: 1 king, 1 queen, 2 rooks, 2 bishops, 2 knights, and 8 pawns.
* The back row holds the major pieces, and pawns fill the row in front. The queen starts on her color (white queen on a light square, black queen on a dark square).

**Piece Movement**:

* **King**: Moves one square in any direction.
* **Queen**: Moves any number of squares in any direction.
* **Rook**: Moves any number of squares horizontally or vertically.
* **Bishop**: Moves any number of squares diagonally.
* **Knight**: Moves in an “L” shape (two squares in one direction and one square perpendicular). Knights can jump over other pieces.
* **Pawn**: Moves forward one square (or two squares on its first move). Captures diagonally.

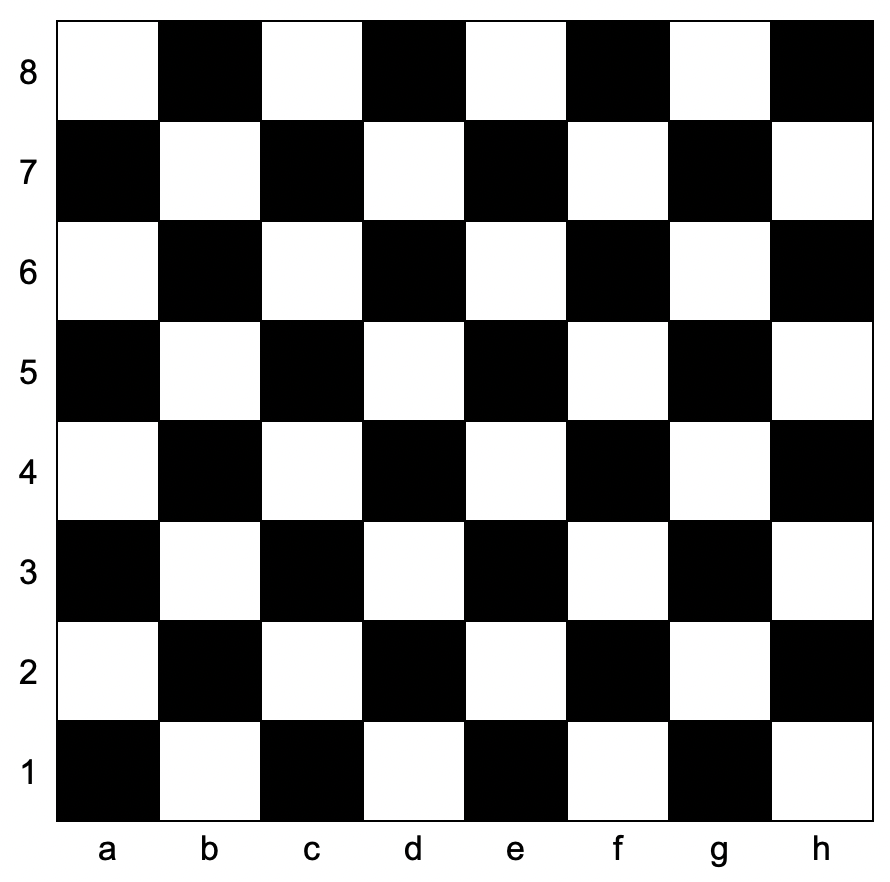
סידור הלוח

* לוח השחמט הוא רשת של 8x8 משבצות בצבעים מתחלפים (שחור ולבן).
* כל שחקן מתחיל עם 16 כלים: מלך, מלכה, 2 צריחים, 2 רצים, 2 פרשים ו-8 רגלים (או חיילים).
* השורה האחורית מכילה את הכלים העיקריים, והרגלים ממוקמים בשורה שלפניה.
* המלכה מתחילה על המשבצת בצבע שלה (מלכה לבנה על משבצת לבנה, מלכה שחורה על משבצת שחורה).

תנועת הכלים

* **מלך**: נע משבצת אחת בכל כיוון.
* **מלכה**: נעה מספר בלתי מוגבל של משבצות בכל כיוון (אופקי, אנכי או אלכסוני).
* **צריח**: נע מספר בלתי מוגבל של משבצות אופקית או אנכית.
* **רץ**: נע מספר בלתי מוגבל של משבצות באלכסון.
* **פרש**: נע בצורת “ר” (שתי משבצות בכיוון אחד ומשבצת אחת בכיוון ניצב). הפרש יכול “לקפוץ” מעל כלים אחרים.
* **רגלי (או חייל)**: נע משבצת אחת קדימה (או שתיים בתור הראשון שלו). רגלי (או חייל) אוכל באלכסון.

Example of a chess board:



**#Statement:** Tell the students: “In our workshop, we will use a 2D array to represent the chess board, where we assume that the size of the board is 8x8, and each coordinate could be:

* “.” = Empty square
* A char that is the first letter of the piece = upper-case is white, lower-case is black.

Just before diving right into the chess workshop, lead a theorethical scanerio:

we are in another dimension, and in our dimension – chess is played with boards sized 9x9, and an empty square is marked by the char “” (epsilon – they should know it from Calculus).

**#Question:** would we need to change our entire code in order for our code to work in the new dimension?

**#Answer:** No!

Talk about creating **final** **class variables** called “BOARD\_SIZE” and “EMPTY\_PLACE”, and how they would be integrated in the functions.

(Create these 2 variables with the students as class variables:

public static final int BOARD\_SIZE = 3;

public static final char EMPTY\_PLACE = '.';)

There are 2 parts to this chess workshop:

**Part 1 – Board validation functions ~30-40 min:**

In this part, we will implement functions that provide board information.  
The solutions to this part’s functions are short and concise.

1. isValidPosition(String pos)
   * Check the length of the input.
   * Extract Row and Column, and check if the row and column are within the range ‘1’ to ‘8’ (valid rows on a chessboard) and ‘a’ to ‘h’ (valid columns on a chessboard) respectively.
2. convertPositionToRowCol(String position)
   * Converts a chessboard position (e.g., “e4”) into row and column indices for array-based representation.
   * Rows are reversed (e.g., ‘1’ maps to 7) and columns map from letters (‘a’ = 0, …, ‘h’ = 7).
3. isWhiteSquare(int row, int col)

* Determines if a chessboard square is white. A square is white if the sum of its row and column indices is even.

1. areTwoSquaresWhite(int row1, int col1, int row2, int col2)

* Checks if both squares are white using the isWhiteSquare function.

1. areTwoSquaresSameColor(int row1, int col1, int row2, int col2)

* Verifies if two squares share the same color by comparing the parity of their row and column sums.

1. isInBounds(int row, int col)

* Ensures that a square lies within valid chessboard boundaries (0 to 7).

1. isWhitePiece(char piece)

* Identify whether a piece is white (PRNBQK).

1. isBlackPiece(char piece)

* Identify whether a piece is black (prnbqk).

**Part 2 – Pieces moves validation functions ~50-60 min:**

In this part, we will implement functions responsible for validating the pieces’ moves.

It might be easier for the students to implement each function only for the black side (the 2 first rows, lowercase letters, where the direction of movement is generally downwards which is positive increment index-wise), and only after that should they re-implement the function to be generic.

1. isValidPawnMove(char pawn, int startRow, int startCol, int endRow, int endCol, char[][] board)

**#Note:** This next function is the most complicated logic move validation function.

You might want to implement this function with the students.

* Validates pawn movement:
  + Moves forward by 1 or 2 squares if on the starting row.
  + Diagonal captures are allowed.
  + Move must not be blocked.

1. isPathClear(int startRow, int startCol, int endRow, int endCol, char[][] board)

* Checks if the path between two squares is unobstructed for straight or diagonal moves.

1. isValidRookMove(int startRow, int startCol, int endRow, int endCol, char[][] board)

* Validates rook movement:
  + Moves straight along rows or columns.
  + Path must be clear, verified by isPathClear.

1. isValidKnightMove(int startRow, int startCol, int endRow, int endCol)

* Validates knight movement:
  + Moves in an “L” shape (2 squares in one direction, 1 in another).

1. isValidBishopMove(int startRow, int startCol, int endRow, int endCol, char[][] board)

* Validates bishop movement:
  + Moves diagonally.
  + Path must be clear, verified by isPathClear.

1. isValidQueenMove(int startRow, int startCol, int endRow, int endCol, char[][] board)

* Validates queen movement:
  + Combines rook and bishop movements.
  + Path must be clear.

1. isValidKingMove(int startRow, int startCol, int endRow, int endCol)

* Validates king movement:
  + Moves 1 square in any direction.

1. isMoveLegal(char[][] board, int startRow, int startCol, int endRow, int endCol)

* Validates a move for any piece type:
  + Ensures the move stays within bounds.
  + Verifies no friendly pieces are at the destination.
  + Calls the right validation function based on the type of piece.

**#Note:** Some of the functions could take some students to a long journey, where in the end these could be solved relatively easily by first thinking about “smart” and “efficient” solution (not naïve or brute-force solutions).

So – try and emphasise the importance of understanding the task, understanding the real-world logic of a chessboard, and translating it into code (and even math) solutions.