1. Rules of the Game

The following section will introduce you to the specific rules of the game. The game is played on a classic 8x8 checker board (some of the exercises might ask you to make this more interesting by offering arbitrary sizes). There are two parties playing the game at any time. One party plays the hounds and the other plays the fox. The hounds try to catch the fox while the fox tries to break through the line of hounds to escape.

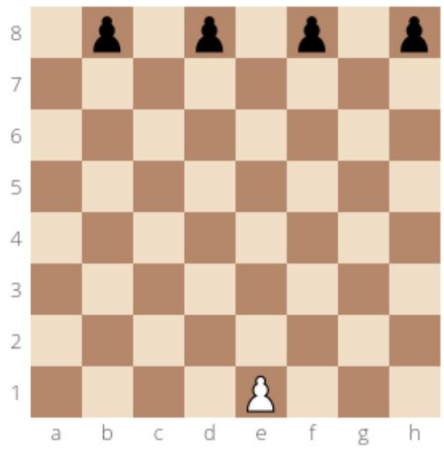

Figure 1: Initial setup of the game board.

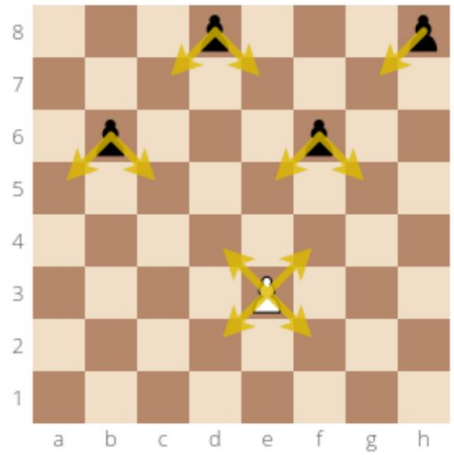

Figure 2: Moving hounds (black) and the fox (white)

You can see the starting setup displayed in Figure 1. In the classic version, there are four hounds lined up on one side of the board and one fox on the other side.

Game pieces can only move on black fields and only one field at a time in a diagonal direction. Hounds can only move forward, while the fox can go in both directions (see Figure 2). Each player can only move one piece per turn and the fox is always the first to move in a new game.

Figures can not move onto each others' fields. As soon as the hounds have blocked the fox into a corner or between themselves so that it can no longer move, they have won. If the fox manages to break through and reaches the other side of the board, it has won.

2. Tasks

The following section contains all tasks you should complete for the assignment.

2.1 Task 1 - Game Setup

In this initial part you should take some time to become familiar with the provided skeleton code. You can execute the given template version by compiling the code and running the main function in FoxHoundGame. There is no need to provide a command line argument at this point. Once you execute it, the application prints which player gets to move next and the main menu (see Listing 1). It then asks you for input in an infinite loop. If you select a 1, the players swap turns but nothing else happens. If you select a 2, the program will terminate.

```
####################################
Fox to move
1. Move
2. Exit
Enter 1 - 2:
```

Listing 1: Main Menu

The state of the game is saved by remembering the positions of all figures on the board and which player gets to move next. Player positions are stored as Strings in an array called players. Their initial positions are supposed to be set in the function **initialisePositions** which can be found in **FoxHoundUtils**. Afterwards, the main game loop is called which handles printing the menu and controlling the game.

2.1.1 Initialising Player Positions

The **initialisePositions** function is not yet implemented. Doing so will be your first job. It takes the board dimensions as parameter, creates a single String array for all figures, fills it with their starting positions and returns the resulting array. In the classic game setup with and 8x8 board, the array will have five entries, one for each hound and one for the fox. By convention, the hounds should always take up the initial positions in the array and the fox the very last. Initialise hound positions from left to right.

All positions should be stored in board coordinates. They consist in a letter indicating the column and a number indicating the row. The initial setup of figures for an 8x8 board should look like shown in Listing 2

```
  ABCDEFGH

1 .H.H.H.H 1
2 ........ 2
3 ........ 3
4 ........ 4
5 ........ 5
6 ........ 6
7 ........ 7
8 ....F... 8

  ABCDEFGH
```

Listing 2: Initial Board Setup

The corresponding players array will contain the following board coordinates:

["B1","D1","F1","H1","E8"]

2.1.2 Variable Board Sizes

Your **initialisePositions** function should work for different board dimensions when calculating the number of players and starting positions. Depending on dimension, you should consider the following rules for placing figures:

- There should be exactly half as many hounds as the value for the board dimension (round up to whole numbers if the dimension is an odd number)
- Hounds should be placed in the first row starting with column B and with one field free in between each. That also means that the top left field is always white.
- There should always be only one fox. Its coordinates are always saved in the last position of the players array
- The fox should always be placed in the last row of the board on a black field as close to the middle column as possible. If the middle is white, place the fox on the black field to its right (see Figure 3 for examples).
- Board dimensions should never be smaller than 4 or larger than 26. This way there are always enough game pieces and you won't have to use double letters for naming columns.

2.1.3 Command Line Arguments

You can test your implementation by running the unit tests or hacking the main method. It should, however, be possible for the user of your application to configure the board dimensions without having to re-compile the code. For this purpose, it should be possible to pass in board coordinates via command line arguments.

The given argument should be between 4 and 26 and assigned to the dimension variable instead of the current default value. If no arguments or invalid command line arguments are provided, you should use the default value.

2.2 Task 2 - Display Game Board

Implement the function **displayBoard** which is currently called in the gameLoop before every print of the main menu. This function should print a graphical representation of the game board and corresponding player positions to the console. It needs information about player positions and the size of the board which are provided via its function parameters

The graphical representation you should print, should look like displayed in Figure 3. Hounds should be indicated with an 'H', the fox with an 'F' and empty fields with a '.' symbol. Note that dimensions above 9 should add a leading zero to row numbers (only in the display, not for board coordinates used throughout the game).

```
  ABCDEFGH

1 .H.H.H.H 1
2 ........ 2
3 ........ 3
4 ........ 4
5 ........ 5
6 ........ 6
7 ........ 7
8 ....F... 8

  ABCDEFGH
```

```
   ABCDEFGHIJK

01 .H.H.H.H.H. 01
02 ........... 02
03 ........... 03
04 ........... 04
05 ........... 05
06 ........... 06
07 ........... 07
08 ........... 08
09 ........... 09
10 ........... 10
11 .....F..... 11

   ABCDEFGHIJK
```

```
  ABCDE

1 .H.H. 1
2 ..... 2
3 ..... 3
4 ..... 4
5 ...F. 5

  ABCDE
```

Figure 3: Different ASCII representations of the game board for a dimension of 8, 11 and 5.

2.3 Task 3 - Fancy Print

Add a function **displayBoardFancy** to **FoxHoundUI** which prints a more fancy version of the game board as you can see here:

```
    A   B   C   D   E   F   G   H
  |===|===|===|===|===|===|===|===|
1 |   | H |   | H |   | H |   | H | 1
  |===|===|===|===|===|===|===|===|
2 |   |   |   |   |   |   |   |   | 2
  |===|===|===|===|===|===|===|===|
3 |   |   |   |   |   |   |   |   | 3
  |===|===|===|===|===|===|===|===|
4 |   |   |   |   |   |   |   |   | 4
  |===|===|===|===|===|===|===|===|
5 |   |   |   |   |   |   |   |   | 5
  |===|===|===|===|===|===|===|===|
6 |   |   |   |   |   |   |   |   | 6
  |===|===|===|===|===|===|===|===|
7 |   |   |   |   |   |   |   |   | 7
  |===|===|===|===|===|===|===|===|
8 |   |   |   |   | F |   |   |   | 8
  |===|===|===|===|===|===|===|===|
    A   B   C   D   E   F   G   H
```

2.4 Task 4 - Moving Figures

Now it is time to move some figures around the board. Double check the possible movement pattern described in Section 1. Figures should be movable by the user of the application. Users should be able to provide a pair of board coordinates to specify which figure to move to which new position. For example, if it is the fox's turn and the board is setup like in Listing 2, the user could input E8 F7 which would move the fox up and to the right.

To allow this, you will have to implement two parts: One is a function which checks if given coordinates constitute a valid move and the second is integrating this with a menu which asks the user for corresponding input.

2.4.1 Move Validity

Let's start with the validity check. Implement a function **isValidMove** in **FoxHoundUtils** which gets five parameters:

- dim An int for the dimension of the board.
- players A String array containing the current player positions.
- figure A char indicating which figure is to be moved ('F' or 'H').
- origin A String indicating the current field of the figure to be moved in board coordinates.
- destination A String indicating the destination field of the figure to be moved in board coordinates.

Make sure this function is public and static so you can call it in the main game loop later on.

The function should then figure out if the specified parameters constitute a valid move and return true if that is the case or false otherwise. You need to consider many things for this check. For example, is the destination field within one diagonal move of the origin, is the origin actually occupied by a figure specified with the given char, are the given destination and origin coordinates actually valid coordinates, is the destination field taken by another figure, etc. Some of those are tested for you with the provided unit tests, others you should figure out yourself.

Considering the initial setup of the board as displayed in Listing 2, have a look at the following examples:

These calls should return **true**:

```
FoxHoundUtils.isValidMove(8, players, 'F', "E8", "D7");

FoxHoundUtils.isValidMove(8, players, 'H', "B1", "C2");
```

These calls should return **false**:

```
FoxHoundUtils.isValidMove(8, players, 'F', "34", "36");

FoxHoundUtils.isValidMove(8, players, 'H', "E8", "D7");

FoxHoundUtils.isValidMove(8, players, 'H', "B2", "A1");
```

2.4.2 Menu Integration

We have provided a flow chart with functionality you should execute when a user inputs a ’1’ in the main menu (see Figure 4). Specifically, you should first ask the user for a pair of coordinates (origin and destination). Check if those coordinates depending on the current placement of the figures is a valid move. If not, print an error message saying "ERROR: Invalid move. Try again!" and ask for coordinates once more.

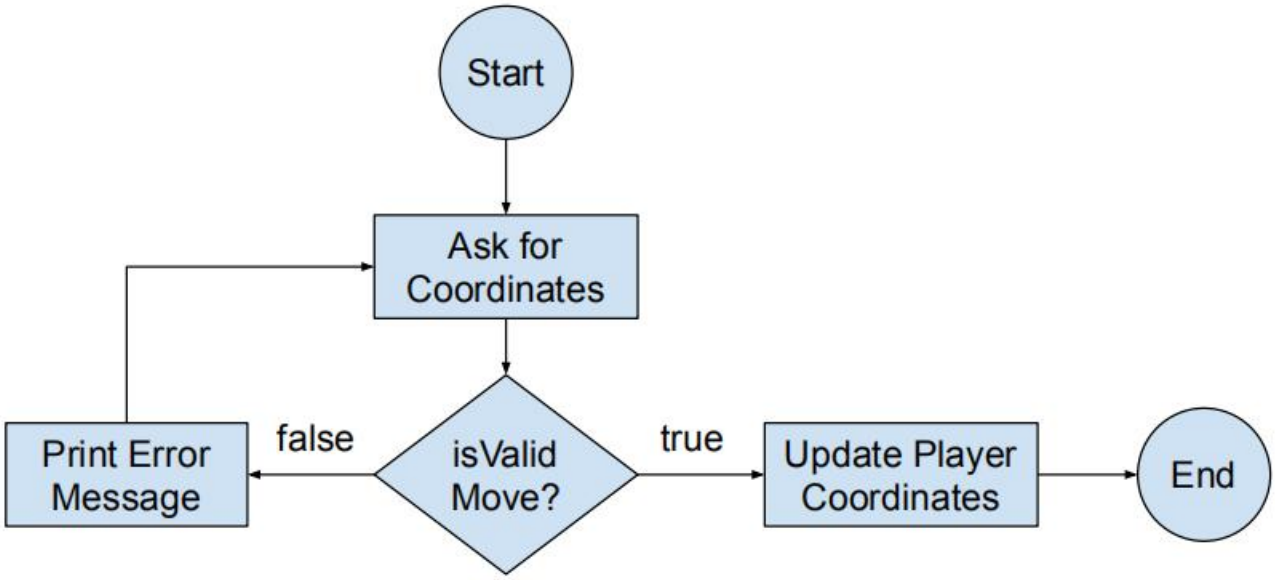

Figure 4: Flowchart for updating player positions including the menu query.

If yes, update the figure’s position in the players array by replacing the old value with the new one for the correct figure.

Asking for coordinates should be done in a similar way to asking for main menu input. For this purpose, implement a function positionQuery in the FoxHoundUI class. This function takes an int parameter for the dimension of the board and a Scanner parameter for reading from stdin. You can use the board dimension here to check if the coordinates given by the player are valid board coordinates. There is no need to check the validity of the move within this particular function. We do that in the calling code as displayed in Figure 4.

The function should print a console output asking the player to input a pair of board coordinates. If the input is a valid pair of board coordinates, you should return a String array6 with two entries: The first contains the origin coordinate and the second the destination. If the input is invalid, print a fitting error message and ask again.

Consider the interaction in Listing 4 as an example. The given input in the first call does not constitute two valid board coordinates, hence an error is printed and the user is asked again. In the second call, two valid coordinates are provided in the correct format. They do not represent a valid move and there will likely be another error message from the calling code, but the positionQuery function is happy.

```
Provide origin and destination coordinates.
Enter two positions between A1-H8:
124 asd
ERROR: Please enter valid coordinate pair separated by space.

Provide origin and destination coordinates.
Enter two positions between A1-H8:
E8 F2
```

Listing 4: Query for a pair of board coordinates.

Once the positionQuery function has returned a pair of coordinates, you should check if they constitute a valid move with your previously implemented isValidMove function. If the move is invalid, print an error message as required above and ask again. If they are valid, update the coordinates of the corresponding figure in the players array.

For example, if it is the fox' s turn to move and the given input is E8 F7, the players array should be updated like this:

Before: ["B1","D1","F1","H1","E8"]

After: ["B1","D1","F1","H1","F7"]

HINT: Reading player positions from the ASCII board is tricky. You might find it useful to write a second function to directly print the current player positions in board coordinates alongside the graphical board.

2.5 Task 5 - Winning the Game

Now that we can move figures around, we need to check if the game is won by any of the parties. For that purpose, implement two new functions: One for checking if the fox won and one for checking if the hounds won.

Implement two functions isFoxWin and isHoundWin in FoxHoundUtils. isFoxWin gets the fox position as String parameter and isHoundWin gets the players array and the dimensions of the board. They check win conditions as specified in Section 1 and return a boolean value indicating if the corresponding figure type has won or not.

Checking for a win should be made directly after a figure has been moved. If the fox wins, print "The Fox wins!" and if the hounds win, print "The Hounds win!". Afterwards, terminate the program.

2.6 Task 6 - Saving and Loading

If everything went well, you should now have a fully functioning game you can play with another person on the same computer. To further exercise your coding skills and deepen your knowledge of the Java language, you will implement two further functionalities for your game. Your game should be able to save the current board to a file and load it back up again.

2.6.1 Saving the Game

Implement a function **saveGame** in the **FoxHoundIO** class. All you need to save the game are the positions of all figures and who has the next turn. This function should therefore get a String array parameter with the player positions, a char indicating which figure will move next and a Path containing the file name for the save game file. Your function should save the game to a file with the given name and return true if saving was successful and false otherwise.

You should save the game state in the following format: In the first line of the save game file, print a single character ’F’ or ’H’ indicating the figure which has the next move in the saved game. This should be followed by all board coordinates of the figures in the game in the same order as they appear in the players array. All entries should be separated by a space character and not terminated with a new line.

You can look at the game files provided with the template to see an example. Also, make sure you function does not override existing files.

2.6.2 Loading a Game

Implement a function **loadGame** in the FoxHoundIO class. This function should get a String array parameter with the player positions and a Path containing the file name for the file to be loaded. Your function should load the file content specified by the given Path, update the players array parameter accordingly8 and return the character indicating the next figure to be moved. If any loading error happened such as an invalid file name was provided, you should return an error code which we indicate by the ’#’ character. In the case of a loading error, the given players array should remain unmodified!

Again, we have provided a few sample files with the template so you can test your function. You can also easily create your own.

2.6.3 Menu Integration

Lastly, we need to make the new functionality available in the main menu. For this purpose, you should extend the main menu with two additional entries. For example:

1. Move
2. Save Game
3. Load Game
4. Exit

If the save or load entries are selected, you will have to ask the user for a file name. Do this by implementing a fileQuery function in FoxHoundUI. This function only needs a Scanner parameter to read the user input from stdIn. It then simply presents a prompt asking the user for the path to a file which it returns in the Path format.

Try to load or save the game with the provided path and evaluate the return value of the corresponding function.

If successful, simply continue the game.If saving failed, print "ERROR: Saving file failed." and continue with the main game loop. If loading failed, print "ERROR: Loading from file failed.", do not change any figures on the board and continue the game.

2.7 Task 7 - OPTIONAL Fox and Hound AI

Should you have finished all of the above, you can try your hand at adding an artificial intelligence for the fox and the hound.

The best way to approach this is to extend the main menu with yet another entry for an AI move instead of a manual one. The game will calculate the corresponding result for whoever's turn it is and update the players array accordingly.

1. Move
2. AI Move
3. Save Game
4. Load Game
5. Exit

To make sure you use the same interface as everybody else, implement the AIs in two different classes:

FoxAI and **HoundAI** which you will have to create yourself. Both classes should get a function called makeMove which receives the players position array and the board dimensions as parameters, calculates a new position, updates the player array parameter accordingly and returns a boolean indicating success or failure.

3. Submission

3.1. All the work is done independently by youself only

NO COPY

Assigned work is to be an individual endeavor. Group or shared assignments will receive 0 points. Discussion about assignments is encouraged, but actual work must be independent.

3.2. Required Files

all java file save in final-src folder.

Programming final Project.docx. (Describe the design part in the report.)

RAR/7Z all required file as final-src.7z

send the file to **eqzhou@163.com**

3.3 Due time

2021-01-08 23:59:59