

# Informatics 1: Object Oriented Programming

## Assignment - Part I


The University of Edinburgh

Volker Seeker (volker.seeker@ed.ac.uk)

### 1 Introduction

For the INF1B course you are expected to complete an assignment which consists in three parts. Once all three parts are completed by the end of the semester, your final course mark will be determined based on your submissions.

This is the first part of the assignment (PART I). Its aim is to help you practice the programming concepts taught during the course and to become familiar with the process of solving assignments. You will have to complete a series of tasks to create a simple Java application for playing a digital board game.

 The programming tasks you will find in this assignment are intentionally challenging to push you to learn the basics well. Make sure you do your lab exercises, attend the tutorials and make the best use of these initial weeks and the support that is offered. Some of the concepts you need to solve these tasks will be taught over the next weeks but do not wait until the last minute to start working on the tasks. Rather attempt them as best you can now and then refine your solutions later.

#### 1.1 Administrative Information

##### Deadline

The assignment is due at **16:00 on 17 February 2020**.

##### Assessment and Feedback

To ease you into the assignment process, PART I is not marked for credit but instead you receive formative feedback to help you tackle PART II and PART III successfully. The feedback you will receive consists in multiple parts:

**Automatic Tests** After the deadline, your submission will be evaluated with a series of automatic tests and a report will be made available to you.


**Feedback Session** A full week of tutorials is dedicated to you discussing your solutions with your tutor should you have any questions.

**Reflective Feedback** PART I is the basis for the second part of the assignment where you will have a chance to reflect on your solution by comparing submissions from other students to each other and to your own.

**Peer Feedback** You are asked in PART II to give actionable advice to your peers about their submissions of PART I which is send out to them afterwards.

**Sample Solution** An official sample solution for you to study will be released after the reflective and peer feedback sections are completed.

Even though this first part is not assessed for credit, we highly recommend that you approach it with your best effort and on your own to be optimally prepared for the following parts. You can review the course's marking criteria in Appendix C.

 Your solution will be shared with other students anonymously in PART II. Please avoid putting information into your code which might allow others to identify you such as uun or author information.

## Coding

For every coding task in the assignment, you are not only expected to write functionally correct code but also maintain high code quality in terms of readability, robustness, documentation, structure and use of the Java language.

Since you have not yet been taught all aspects of the course and to make the feedback process more streamlined, please consider the following restrictions for your implementations:

- Put all of your implementations into the provided four .java files only and do not implement any new classes.
- Do not use any third party libraries, **only the Java API** (see Appendix B for some useful suggestions).
- Do not use any collection classes such as lists or maps to solve the given tasks.
- Do not use any functional language constructs such as lambdas or streams (that was last semester).
- Keep all classes in the default package, i.e. no package. This will allow the automatic tests to run through smoothly.

If you already know some or all of the aspects mentioned above, we recommend that you consider it a challenge to yourself to follow the restrictions for solving the given tasks.

**Write Robust Code** Make sure you write robust code that handles potential error cases as discussed in the *Testing and Debugging* lecture. As a guideline regarding INF1B and unless otherwise stated, invalid input to public methods should throw `IllegalArgumentException`s with expressive error messages or `NullPointerException`s for null references. This should, of course, be documented for the corresponding method.

**Write Readable Code** Make sure you follow the coding conventions for the INF1B course we have specified during the *Java API and Documentation* lecture.

## Submission

The submission should happen via the CODEGRADE infrastructure. You can find more information on how to use it and what to submit in Appendix A.

Before the deadline, you can and should submit as often as possible. Your latest submission will then always be replaced with the newest version. Some of the auto tests will run for every submission you put in. This way, you can get an estimate of how well you are doing early on and whether your submission was successful. The full set of tests will be executed for your final submission after the deadline.

👉 Please be aware that there is **no deadline extension** possible for this part of the assignment to allow a smooth progression of the course. If you miss the deadline, it won't affect your final mark but significantly inhibit your own learning process.

## Support

When solving PART II and PART III for credit, you will have to adhere to the good scholarly practice clause which we have included below. To be optimally prepared, we recommend that you follow the same procedure for this part of the assignment. That means, you should not share any code with your fellow students.

Nevertheless, you will have the possibility to get support should you need it. At any time, you are welcome to discuss the assignment with any course instructors. You can do that during teaching sessions or via the course's [Piazza forum](#)<sup>1</sup>.

When posting on Piazza, please put all questions into the corresponding assignment folder. Feel free to discuss general techniques amongst each other unless you would reveal an answer. If in doubt about revealing an answer, please ask the instructors privately.

Lastly, please make good use of any internet or literature resource you consider helpful for solving the tasks (that, of course, includes all the course material released so far). This is how you would normally write code anyway and we strongly encourage you to make use of this practice.

## Good Scholarly Practice

Please remember the University requirement as regards all assessed work for credit. Details about this can be found at:

<http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct>

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

---

<sup>1</sup><https://piazza.com/ed.ac.uk/spring2020/inf1b>

## 2 Content

This section will give you an overview over the content of the actual tasks. You will build a game called Fox and Hounds which is played on a classic 8x8 chess board. Next to game elements such as figures and game logic such as moving figures and checking win conditions, you will also implement a command line interface to display and control the game state as well as functionality for loading and saving games.

### 2.1 Template Material

**Skeleton Classes** To get your started with the implementation, four skeleton classes are provided. Those four classes are the only ones you need to work on and ultimately submit. The four classes are `FoxHoundGame`, `FoxHoundUtils`, `FoxHoundUI` and `FoxHoundIO`. Please read through the classes' documentation for more details on their individual jobs.

**JUnit Tests** Similar to lab exercises, you are provided with a set of JUnit tests which will help you check the functional correctness of your implementations. We highly recommend that you test your code vigorously after each new change.

Be aware, that the provided tests do not offer complete functional coverage of all possible inputs to your program. The automatic tester will use the provided tests as well as additional advanced tests for checking corner cases and specific scenarios you should think of on your own. For that purpose, feel free to extend the provided tests or hack the given main function.

**Game Files** You are given a set of Fox and Hound game files with different board setups. This will aid you when testing the load and save functionality later on.

### 2.2 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.

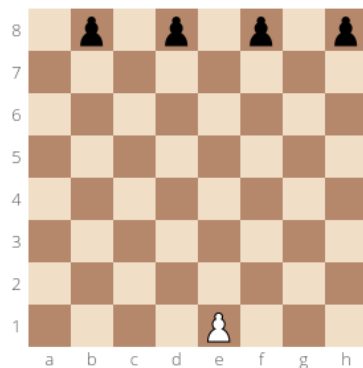


Figure 1: Initial setup of the game board.

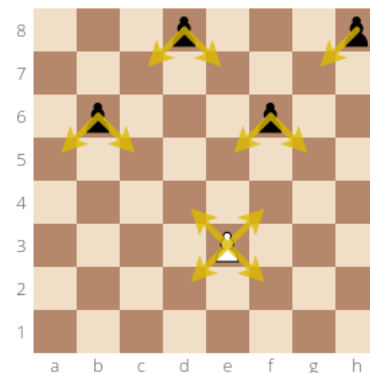


Figure 2: Moving hounds (black) and the fox (white)

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.

You can find another more detailed description of the rules including a video with various strategies in this [article](#)<sup>2</sup>.

### 3 Tasks

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

Some parts are marked as **ADVANCED** which means they are usually more difficult to solve but not necessarily required for a passing solution. You might find that some self study of Java language features and API classes is required to tackle them.

Some parts are marked as **OPTIONAL** and are not considered when marking the code. Feel free to attempt them for an additional challenge and to further improve your coding skills.

#### Task 1 - Game Setup


◀ Task

In this initial part you should take some time to become familiar with the provided skeleton code. It might contain the occasional pointer towards implementing later tasks.

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.

```
1 #####
2 Fox to move
3
4 1. Move
5 2. Exit
6
7 Enter 1 - 2:
```

Listing 1: Main Menu

 The main game loop makes use of a Java `switch` statement which is a convenient way of handling multiple conditions of a variable. It can be replaced by an `if else` construct which is, however, often more difficult to read (An even better way to handle a menu such as this would be a *Command Design Pattern* which requires advanced OO knowledge). Have a look at this [tutorial](#)<sup>a</sup> for more information regarding `switch`.

<sup>a</sup><https://www.geeksforgeeks.org/switch-statement-in-java/>

The state of the game is saved by remembering the positions of all figures on the board and which player

<sup>2</sup><https://chessplus.net/foxandhounds/>

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.

## 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 an 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<sup>3</sup>.

1	ABCDEFGH
2	
3	1 .H.H.H.H 1
4	2 ..... 2
5	3 ..... 3
6	4 ..... 4
7	5 ..... 5
8	6 ..... 6
9	7 ..... 7
10	8 ....F... 8
11	
12	ABCDEFGH

Listing 2: Initial Board Setup

The corresponding `players` array will contain the following board coordinates:

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

## ADVANCED Variable Board Sizes

Since creating an array with those coordinates is trivial, we are going to make this more interesting. 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.

<sup>3</sup>Note that row numbers are reversed compared to Figures 1 and 2

- 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.

👉 Seriously consider which helper functions might be useful when calculating board coordinates. They will most likely become relevant and very helpful for later tasks.

**HINT:** If you have trouble solving this task or the following tasks for varying dimension, start with solving them for the default dimension only and extend for a variable dimension afterwards.

📝 When putting the board coordinates together, you should consider working with a char type<sup>a</sup>. Chars can be treated like numbers and easily cast to integers. For example, the following Java construct prints `true`.

```
1 char value = 'B' + 2;
2 System.out.println(value == 'D');
```

Listing 3: Char Examples

<sup>a</sup>One of the lab exercises in week 4 will give you some more practice using chars.

## 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.

## Task 2 - Display Game Board

◀ Task

The next requirement for a functional game is a graphical output of the current state of the game board. The easiest way to achieve this is by using ASCII graphics<sup>4</sup>.

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).

<sup>4</sup>[https://en.wikipedia.org/wiki/ASCII\\_art](https://en.wikipedia.org/wiki/ASCII_art)

```

                                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

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

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

                                ABCDE
                                ABCDEFGHIJK

```

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

### Task 3 - OPTIONAL Fancy Print

◀ Task

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

```

### Task 4 - Moving Figures

◀ Task

Now it is time to move some figures around the board. Double check the possible movement pattern described in Section 2.2. 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.

## 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");
```

 **Error Handling:** For this function, you should make a difference between invalid input and invalid moves. Invalid moves should result in a return value of `false`. For example, if you attempt to move the fox but give the coordinates of a field occupied by a hound. Invalid input should result in the corresponding exception (see Section 1.1). For example, if the given players array is `null` or the figure parameter not 'F' or 'H'.

## Menu Integration

Now that we have the checkup done, let's integrate this into our main loop and actually move the players.

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.

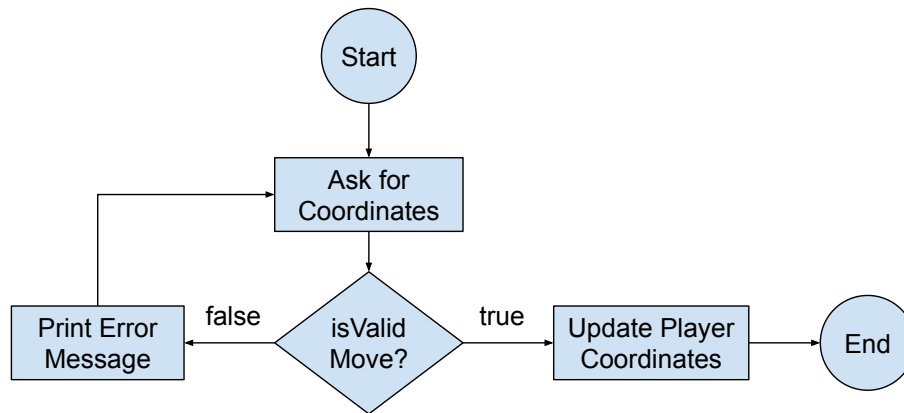


Figure 4: Flowchart for updating player positions including the menu query.

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<sup>5</sup>.

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` array<sup>6</sup> 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.

```

1 Provide origin and destination coordinates.
2 Enter two positions between A1-H8:
3 124 asd
4 ERROR: Please enter valid coordinate pair separated by space.
5
6 Provide origin and destination coordinates.
7 Enter two positions between A1-H8:
8 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.

<sup>5</sup>Handling the UI and handling the game logic should be uncoupled. This way, we can change the underlying logic or the display of the board without affecting the other.

<sup>6</sup>Using OO, this can be made much more type safe and we will show you later how.

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.

## Task 5 - Winning the Game

◀ Task

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 2.2 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<sup>7</sup>. If the fox wins, print "The Fox wins!" and if the hounds win, print "The Hounds win!". Afterwards, terminate the program.

## Task 6 - ADVANCED Saving and Loading

◀ Task

If everything went well, you should now have a fully functioning game you can play with another person on the same computer. Admittedly, it is likely more tedious than playing on an actual board but you implemented it all yourself!

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.

This task requires you to make extensive use of Java API classes and handle potential exceptions. Consider the suggestions in Appendix B for some inspiration and make good use of the internet to find out how to use them.

👉 **Error Handling:** Similar to the `isValidMove` function, the save and load functions should throw exceptions for invalid input such as the players array being `null`. Errors during saving or loading, however, should be handled gracefully by presenting an error message and returning an error value or `false` to indicate a failed attempt.

👉 To make this a bit easier, you can work with the following assumption:


- only classic 8x8 boards can be saved and loaded
- if the file content is correctly formatted, it will always contain a valid positioning of figures

## 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

<sup>7</sup>We assume that the initial position of the board will never contain a winning configuration of figures.

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.

 The `Path` class is a programmatic representation of a path in the file system. Have a look at this [article<sup>a</sup>](https://examples.javacodegeeks.com/core-java/nio/file-nio/path/java-nio-file-path-example/) for more information and how to use it.

<sup>a</sup><https://examples.javacodegeeks.com/core-java/nio/file-nio/path/java-nio-file-path-example/>

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 your function does not override existing files.

## 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 accordingly<sup>8</sup> 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.

## 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.

---

<sup>8</sup>Side effects of functions and arrays as reference types are discussed during the lectures.

## Task 7 - OPTIONAL Fox and Hound AI

◀ Task

Should you have finished all of the above, you can try your hand at adding an artificial intelligence for the fox and the hound. You can then play against the computer, pitch both against each other or swap your AI code (**AND ONLY THAT CODE!!**) with a fellow student and see if they or their AI can beat it.

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.

You might notice that writing an intelligent AI can be quite tricky without saving additional data between each call. Since this is an optional task, all the restrictions from Section 1.1 shall not apply here.

## Appendix A Submission

We are using an online tool called CODEGRADE to automatically grade your code for correctness and robustness. You can access this tool via the INF1B [Learn page](#)<sup>9</sup> by following the menu:

[Assessment] » [Assignment Part I] » [Assignment Part I - Submission].

This will open the CODEGRADE web interface for you within the Blackboard environment. Here you have the options to review a previous submission and the feedback you received so far, hand in a new submission (which will override the previous one) or see the grading rubrics for this auto test.

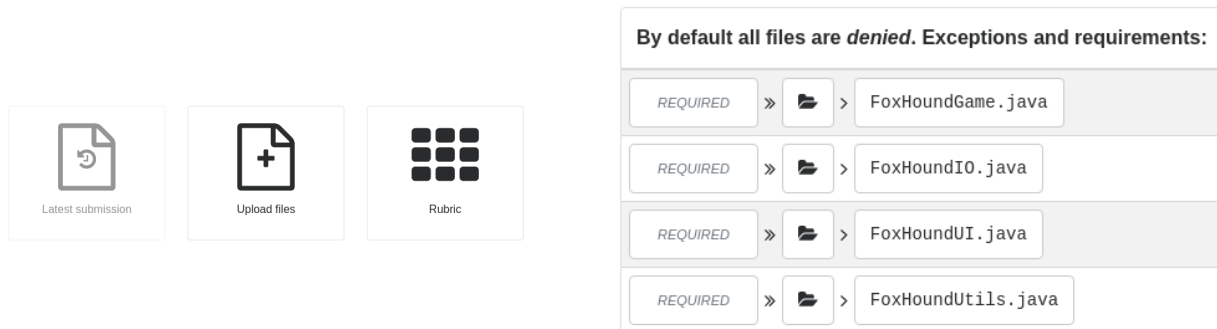





Figure 5: CODEGRADE menu for submission and hand-in instructions.

You are required to hand in the four files you were given with the assignment template. Please put all your solutions into those four files according to the specifications in Section 1.1. You can submit individual files or archives such as zip or tar.

You can find a more detailed description on the [CODEGRADE website](#)<sup>10</sup>. We are not using Git or Group submissions for our assignments.

 One of the initial criteria checked for your submission is that all files compile together with the provided basic unit tests. Make sure this step runs through. If you did not manage to solve all exercises, provide required function dummies to enable compilation.

 We recommend that you try the submission process as early as possible (even with a partial or dummy solution) to make sure everything works as expected. Keep in mind that there is **no deadline extension** for this part of the assignment.

 Keep in mind that this auto grader only looks at one category considered for the final marking process which is *Correctness and Robustness*. Although this is an important part, it is not the only aspect we consider when putting your final course mark together. Have a look at the marking criteria in Appendix C for more details.

<sup>9</sup><https://course.inf.ed.ac.uk/inf1b>

<sup>10</sup><https://docs.codegra.de/guides/use-codegrade-as-a-student.html>

## Appendix B Useful Java Classes

Good code should not reinvent the wheel. Even though you are asked not to write your own classes or use collections for this part, there are many other Java classes in the API which you can and should use to make your life easier and increase your code quality.

Here is a sample of classes which might be helpful when solving the given tasks:

- `java.lang.Math`
- `java.lang.String` and its many functions
- `java.lang.StringBuilder`
- `java.util.StringJoiner`
- `java.util.Scanner`
- `java.util.Random`
- `java.util.Arrays`
- `java.util.Objects`
- `java.io.PrintWriter`
- `java.io.File`
- `java.io.Files`
- `java.nio.file.Path`
- `java.nio.file.Paths`
- `java.util.Objects`

Check out the [Java 11 API pages](https://docs.oracle.com/en/java/javase/11/docs/api/index.html)<sup>11</sup> for more information.

---

<sup>11</sup><https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

## Appendix C Inf1B Marking Criteria

This section contains a description of the assessment scheme used to determine the overall course mark for each student at the end of the semester. This happens after you have submitted the third part of your assignment at the end of week 11. For this process we will consider the feedback you generated for PART II and your code submission for PART III.

We will consider the following criteria:

**Completion** Those with less previous experience may have difficulty completing all of the assignment tasks. It is possible to pass without attempting the more advanced tasks - and a good solution to some of the tasks is much better than a poor solution to all of them.

**Readability & Code Structure** Code is a language for expressing your ideas - both to the computer, and to other humans. Code which is not clear to read is not useful, so this is an essential requirement.

**Correctness & Robustness** Good code will produce “correct” results, for all meaningful input. But, it will also behave reasonably when it receives unexpected input.

**Use of the Java Language** Appropriate use of specific features of the Java language will make the code more readable and robust. This includes, for example: iterators, container classes, enum types, etc. But the structure of the code, including the control flow, and the choice of methods, is equally important.

**Code Review** Working with code provided by other developers is a major part of working with software. The ability to assess a piece of code written by someone else and to provide meaningful feedback on its quality is therefore an essential skill you should learn.

Marks will be assigned according to the [University Common Marking Scheme](#)<sup>12</sup>.

The following table shows the requirements for each grade. All of the requirements specified for each grade must normally be satisfied in order to obtain that grade.

### Pass: 40-49%

- Submit some plausible code for a significant part of assignment PART III, even if it does not work correctly.
- Demonstrate some understanding of the issues involved in your answers to at least one question in the Code Review from PART II.

### Good: 50-59%

- Submit working code for most parts of assignment PART III, even if there are small bugs or omissions.

---

<sup>12</sup><https://web.inf.ed.ac.uk/infweb/student-services/ito/students/common-marking-scheme>



- Submit code which is sufficiently well-structured and documented to be comprehensible. This includes an appropriate use of Java features and choice of methods, as well as layout, comments and naming.
- Demonstrate some understanding of the issues involved in your answer to all of the questions in the Code Review from PART II.

**Very Good: 60-69%**

- Submit working code for all parts of assignment PART III, even if this contains small bugs.
- Submit code which is well-structured and documented.
- Demonstrate good understanding of the issues involved in your answers to all of the questions in the Code Review from PART II and provide plausible and actionable solutions for some.

**Excellent: 70-79%**

- Submit and demonstrate working code for all parts of assignment PART III, with no significant bugs.
- Submit code which is well-structured and documented.
- Demonstrate good understanding of the issues involved in your answers to all of the questions in the Code Review from PART II and provide plausible and actionable solutions for all of them.

**Excellent: 80-89%**

- Marks in this range are uncommon. This requires faultless, professional-quality design and implementation, in addition to well-reasoned and presented answers to the Code Review questions.

**Outstanding: 90-100%**

- Marks in this range are exceptionally rare. This requires work “well beyond that expected”.