CS1002 – Object Oriented Programming

Assignment: **W09 – Fox and Geese**

Deadline: 17 Nov 2017 *Credits*: 25% of coursework mark

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Aim / Learning objectives

Your aim in this practical is:

- to use 2-D arrays as a data structure in Java programs

By the end of this practical you should be able to:

- design and implement data structures based upon 2-D arrays
- design and implement methods that manipulate 2-D arrays
- carry out appropriate tests to confirm your program's functionality

Introduction

This practical involves modelling and implementing a solution to a specified problem. You will need to identify which classes to create, what fields they will have and which methods they will need. You will also need to implement a solution and test it.

The main, compulsory, part of the practical is described in Parts 1-6. Part 7 describes some additional extension activities, which you can choose to do or not. Without doing any extension activities the highest mark that you can achieve is 17. As before, the quality and design of the code and report is extremely important for getting high marks.

Setting up

The first steps should be familiar:

- Log in to a machine booted into *Linux*.
- Check your email in case of any late announcements regarding the practical.
- Launch *Terminal*. Use appropriate commands to create a new directory called *W09-Practical* in your *cs1002* directory and move to it.
- Create a new *LibreOffice Writer* document for your report, add the appropriate header at the top, and save it as *W09-Practical-Report*.
- Create a *source* directory inside the *W09-Practical* directory and move into it. This is where your program source code should reside.
- Copy the starter code from the practical directory on studres into this directory.

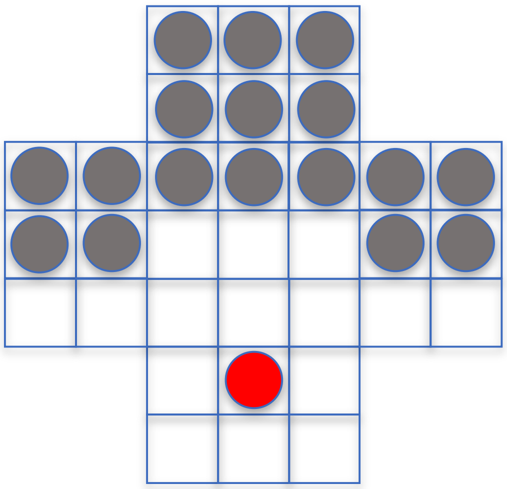
If you can't remember how to do these steps, refer back to the instructions for the Week 1 practical at:

https://studres.cs.st-andrews.ac.uk/CS1002/Practicals/W01/
https://studres.cs.st-andrews.ac.uk/CS1002/Lectures/

Overview

In this practical you will design and implement a program to simulate the board game Fox and Geese. There are several variations on this game, so it is important to read the following description carefully **– this is the variant you will be implementing.**

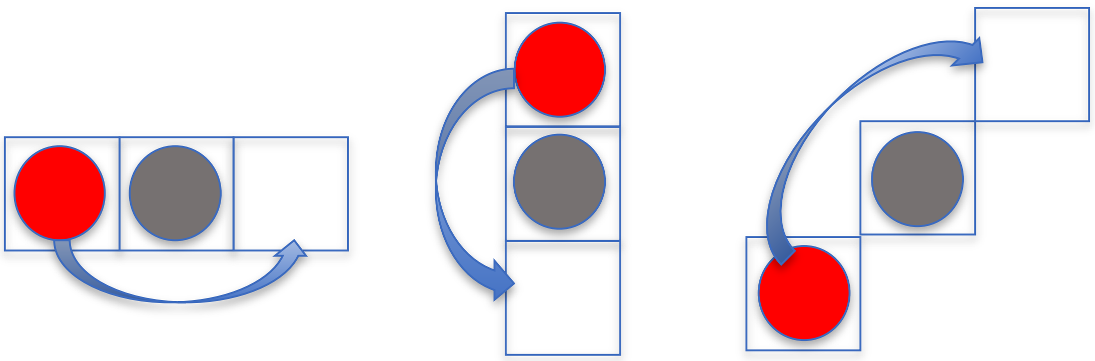
The Fox and Geese board is in the shape of a cross, as follows:



Board and Pieces
The board consists of thirty-three squares, nine in the centre, and six more in each of the arms of the cross. There is a single piece representing the fox (shown as a single red piece in the above diagram), and a number of pieces representing the geese (shown as grey pieces above it). In the variant of the game that we are modelling, we will assume **17 geese**, as in the figure above. The starting positions of these pieces are also shown in the figure.

Players, Movement, and Capture
One player controls the fox, the other the geese. The geese move first. The players take turns to move the fox or **one** of the geese to an adjacent unoccupied square, horizontally, vertically, or diagonally.

The fox may alternatively capture a goose by jumping over it (in a straight line) onto an unoccupied square:



A captured goose is removed from play for the remainder of the game.

Victory Conditions
The geese win the game if the fox is unable to move. This happens when the fox is surrounded by geese so that there are no adjacent squares to which it can move, and no geese that it can capture. The fox wins the game when there are no geese left on the board.

It might be useful to play a few games with a friend to get a feel for the game before starting work.

Design and Implementation

Specification
The task in this practical is to write a program to support the playing of a game of Fox and Geese. It would be sensible to review your lecture notes on Noughts & Crosses before continuing. Fox and Geese shares many of the same design considerations. You will need to think about the following:

- **Board representation**. The practical specifies a two-dimensional array. Since the board is not rectangular, your array will have to be big enough contain the whole board and you will also have to decide how to represent a position outside the board.
- **Next to Play**. How will you keep track of whose turn it is?
- **Making moves**. How will you specify a move (**hint**: one pair of coordinates, or two?). How will you ensure that a specified move is legal? How will you update the state of the board to reflect that a move has been made?
- **Victory**. How will you detect victory from the perspective of the fox? From the perspective of the geese?

Design and formatting
There is starter code with some Java classes provided with this specification. You should use it to help you get started, but feel free to change and extend it as needed as long as it conforms to the specification explained below, which is used by the autochecker.

The game should start by printing the board on the terminal using a textual representation. Each free square is marked by a dot (.), each goose by a lowercase 'o', and the fox by an asterisk (*). The provided Board class contains a method which does this.

It should then state whose turn it is to play, prompt the user to enter a move, and then read four integers from the user: the x and y coordinates of the starting square and the x and y coordinates of the target square. All coordinates start at 0. If the move is not allowed, it should print "Illegal move!", display the board and ask again. If the move is legal, it should update the board, display the new state of the board and prompt the user again. A sample game is shown below (the input typed by the user is shown in red):

```
  ooo
  ooo
ooooooo
oo...oo
.......
  .*.
  ...
Geese play. Enter move:
0
3
0
4
  ooo
  ooo
ooooooo
.o...oo
o......
  .*.
  ...
Fox plays. Enter move:
0
0
1
2
Illegal move!
  ooo
  ooo
ooooooo
.o...oo
o......
  .*.
  ...
Fox plays. Enter move:
```

The first user input moves a goose from square (0,3) to square (0,4). This is a legal move and is therefore accepted. The goose is moved, the new board displayed, and the fox is next to move. The second user input tries to move the fox from square (0,0) to square (1,2). Since the starting square does not contain a fox (and is outside of the board), the move is not accepted and fox is still next to play.

The game should continue until one side wins. Then a message is displayed (e.g. "Fox wins!") and the program terminated. To stop a game at any time, press CTRL+C on the keyboard.

All the messages are defined as String constants in the starter code, to make it easier to match the expected output. For example, to print the victory message for the fox, you could write

```
System.out.println(FOXWINS_MSG);
```

From any method inside the Game class. Have a look at the provided starter code for more examples.

Testing and the Autochecker

Like most previous practicals, this assignment will make use of the School's automated checker. You should therefore ensure that your program can be tested using the autochecker. It should help you see how well your program performs on the tests we have made public and will hopefully give you an insight into issues prior to submission. The automated checking system is simple to run from the command line in your *W09-Practical* directory:

```
stacscheck /cs/studres/CS1002/Practicals/W09/Tests
```

Make sure to type the command exactly – occasionally cutting and pasting from the PDF spec will not work correctly. If you are struggling to get it working, ask a demonstrator.

The automated checking system will only check for basic operation. It is up to you to provide evidence that you have thoroughly tested your program. One possible way of testing would be to play a number of games through, testing legal and illegal moves, as well as different victory conditions to see if everything worked correctly.

At each step, you could try one/some of the following:

- **A legal move.** This would change the state of the game, and result in the fox or one of the geese changing its position, or being removed from the board. This should be indicated.
- **A move outside one's turn.** A fox should not be allowed to move if it's the geese's turn to move, and vice versa.
- **An illegal move.** A move should not be possible if a cell is already occupied or outside the board, and should be indicated.
- **A move that leads to victory.** Victory should be indicated immediately after a move that results in it.

Report and Upload

Your report **must** be structured as follows:

- **Overview**: Give a short overview of the practical: what were you asked to do?
- **Design**: Describe the design of your solution, justifying the decisions you made**.** In particular, describe the classes you chose, the methods they contain, a brief explanation of why you designed your model in the way that you did, and any interesting features of your Java implementation.
- **Testing**: Describe how you tested your program and in particular, how you designed different tests. Your report should include the output from a number of test runs to demonstrate that your program satisfies the specification.
- **Evaluation**: Evaluate the success of your program against what you were asked to do.

- **Conclusion**: Conclude by summarising what you achieved, what you found difficult, and what you would like to do given more time.

Don't forget to add a header including the practical name, your matriculation number, tutor, and the date.

Package up your *W09-Practical* folder and a **PDF** copy of your report into a zip file as in previous weeks, and submit it using MMS, in the slot for Practical W09.

Extension Activities

The activities in this section are not compulsory, though you need to do at least one of them to achieve a grade above 17. Try them if you're interested and have spare time.

- Can you replace one or both of the human players with an artificial intelligence? The very simplest such AI would collect all of the possible moves together and select one at random, but you can do much better than that…
- Can you support arbitrary board sizes? How would you initialise the board in this case?
- Can you support some of the variants of the game, such as an increased number of foxes, or a different number of geese? How will you allow the user to select which game to play?

You are also welcome to come up with extensions of your own if you are feeling adventurous. Any such extension should be related to the main topic of the practical, namely implementing a board game using arrays. As before, make sure to submit your extensions as new classes, so they do not interfere with the autochecker.

Please note

Assessment Criteria

Marking will follow the guidelines given in the school student handbook (see link in next section). Some specific descriptors for this assignment are given below:

Mark range	**Descriptor**
1 - 6	Very little evidence of work, software does not compile or run, or crashes before doing any useful work. You should seek help from your tutor immediately.
7 - 10	A decent attempt which gets part of the way toward a solution, but has serious problems such as not compiling, or crashing often.
11 - 13	A solution which is mostly correct, but has major issues such as allowing incorrect moves, not being able to complete a game, has poor readability, occasionally crashes due to poor bound checking, or is accompanied by a weak report riddled with mistakes.
14 - 15	A mostly correct solution accompanied by a good report, but which can be improved in terms of code quality, for example: poor method decomposition, lack of comments, hard-coded initialisation instead of loops, or overly complex implementation.
16 - 17	A correct solution which demonstrates excellent design and code quality, good method decomposition and comments, good error checking and testing, accompanied by a well-written report.
18 – 19	As above, but implementing one or more extensions, accompanied by an excellent report. Submissions in this range must have excellent code quality, and excellent report, and demonstrate extensive testing.
20	As above, but with multiple extensions, outstanding code decomposition and design, and accompanied by an exceptional report.

Policies and Guidelines

Marking
See the standard mark descriptors in the School Student Handbook:
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness penalty
The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good academic practice
The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/students/rules/academicpractice/