University of St Andrews
School of Computer Science

CS1002 – Object Oriented Programming

Assignment: **W09 – Simplified Chess**

Deadline: 16 Nov 2018 *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 1-5. Part 6 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 and you are strongly encouraged to produce a complete, well-tested and well-documented solution before attempting extensions.

1. 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/

2. Overview

In this practical you will design and implement a program to simulate a simplified form of chess. If you are unfamiliar with the game you should start by reading the description on Wikipedia:

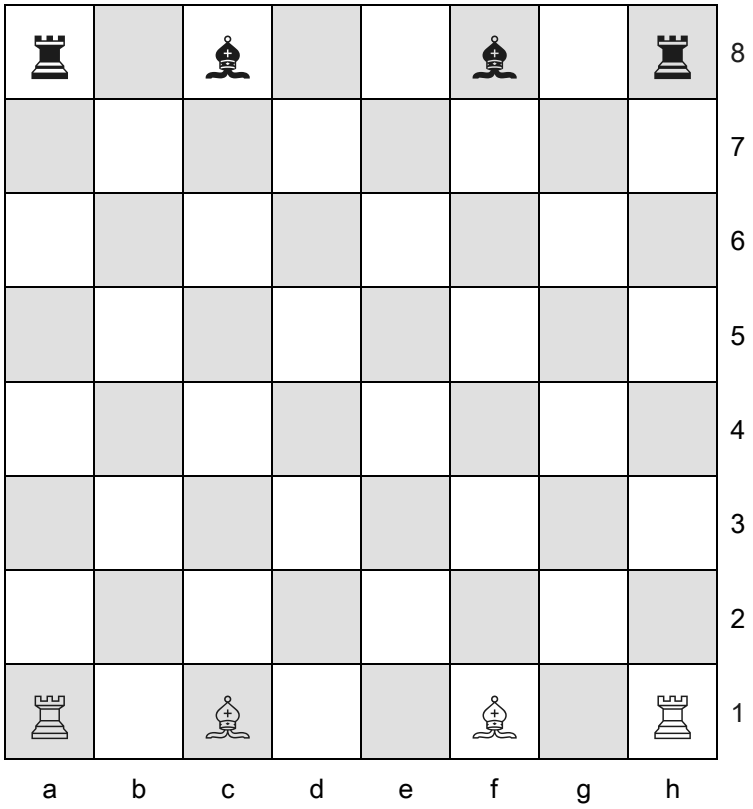
http://en.wikipedia.org/wiki/Chess

Pay particular attention to the way in which the different pieces move.

We define *Simplified Chess* for the purposes of this practical as being played on an 8 x 8 board, where white and black both have two bishops and two rooks (castles). The game is over when one player has lost all of his/her pieces. The task of the practical is to design and implement a program to:

a) represent the current state of a game of simplified chess,
b) allow this state to be updated by making moves, and
c) recognise that the game is over and disallow further moves from being made.

The initial state of the board should be as follows, with **white** to move first:



When run, your program should display the board (this functionality is already implemented in the starter code), state whose turn it is, and ask the user to enter the next move. This can be accomplished by using the supplied **EasyIn** library to read in two strings from the user. The move is represented by the start and end coordinates using the coordinate system on the board above. For example moving the white rook from position "a1" to position "a4" would be accomplished by the following interaction:

```
White plays. Enter move:
a1
a4
```

If the move specified is **legal** (correct colour for whose turn it is to play, and a legal move for a specified piece), the program should **update the board state** and **display the board**. If the move is illegal it should display an error message and wait for the next command. If the game is over, it should reject the move and display an appropriate message. If the user enters "quit" at any point, the game should immediately terminate.

You are expected to employ a 2-D array to represent the chess board and to decompose your solution into **classes and methods**, with a very basic **main** method which starts a game. The starter code on studres contains skeleton code which should help you with your design. The starter code also implements a method for printing the board, as well as all expected messages. You are encouraged to use these in order to comply with the autochecker.

3. Design and Implementation

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

- **Board representation**. The practical specifies a two-dimensional array. The board is square and your array will have to be big enough to contain the whole board.
- **Next to Play**. How will you keep track of whose turn it is?
- **Making moves and updating the board**. How will you convert the positions from a string (e.g. "a2") to integer indices that can be used with an array (e.g. 0 and 1)?
- **Checking moves**. 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?

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 (.), and each of the pieces by a special Unicode character. The provided Board class contains a method which does this. (**Hint**: the pieces might have inverted colours if your terminal uses a dark background (i.e. white pieces will appear to be black and vice versa). Switch to a bright terminal background if you find this confusing. Increasing font size in the terminal will make the board easier to see.)

Your program should then state whose turn it is to play, prompt the user to enter a move, and then read two strings from the user: the starting and the ending position (e.g.: "c3" or "d7"). 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):

```
  a b c d e f g h
8 ♜ . ♝ . . ♝ . ♜ 8
7 . . . . . . . . 7
6 . . . . . . . . 6
5 . . . . . . . . 5
4 . . . . . . . . 4
3 . . . . . . . . 3
2 . . . . . . . . 2
1 ♖ . ♗ . . ♗ . ♖ 1
  a b c d e f g h

White plays. Enter move:
c1
e3
```

```
  a b c d e f g h
8 ♜ . ♝ . . ♝ . ♜ 8
7 . . . . . . . . 7
6 . . . . . . . . 6
5 . . . . . . . . 5
4 . . . . . . . . 4
3 . . . . ♗ . . . 3
2 . . . . . . . . 2
1 ♖ . . . . ♗ . ♖ 1
  a b c d e f g h

Black plays. Enter move:
c8
b5
Illegal move!
  a b c d e f g h
8 ♜ . ♝ . . ♝ . ♜ 8
7 . . . . . . . . 7
6 . . . . . . . . 6
5 . . . . . . . . 5
4 . . . . . . . . 4
3 . . . . ♗ . . . 3
2 . . . . . . . . 2
1 ♖ . . . . ♗ . ♖ 1
  a b c d e f g h

Black plays. Enter move:
```

The first user input moves the white bishop from position c1 to position e3. This is a legal move and is therefore accepted. The bishop is moved, the new board displayed, and black is next to move. The second user input tries to move the black bishop from position c8 to position b5. This is not a legal move for a bishop (as bishops can only move diagonally), so the move is not accepted and black is still next to play.

The game should continue until one side wins. Then a message is displayed (e.g. "White wins!") and the program terminated. To stop a game at any time, type "quit" at the prompt.

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 white player, you could write

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

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

4. Testing and the Autochecker

Like most previous practicals, this assignment will make use of the School's automated checker "stacscheck". 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 one of the pieces changing its position, or being removed from the board. This should be shown on the screen.
- **A move outside one's turn.** A white piece should not be allowed to move if it's black's turn to move, and vice versa.
- **An illegal move.** A move should not be possible if a cell is already occupied by a piece of the same colour, or if the target cell is outside the board.
- **A move that leads to victory.** Victory should be indicated immediately after a move that results in it.

5. 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. Please note that simply reporting the result of stacscheck is not enough; you should do further testing and explain in the report why you think that your testing is sufficient to prove that your program is correct.
- **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.

6. 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.

- Update your program to support the **queen** (HINT: a queen is like a rook and bishop combined) and **knight** pieces.
- Update your program to support the king piece. In order to do so you will need to support the concept of "check" – see the description at the web page above. Can you now modify the game over condition to be "checkmate" as in real chess? What about "stalemate"?
- 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?

You are also welcome to come up with extensions of your own if you are feeling adventurous (e.g. implementing a complete game of chess, including pawns and promotion of pawns to other pieces). 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, use of magic literals instead of constants, 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 at least one extension, accompanied by an excellent report. Submissions in this range must have excellent code quality, include an excellent report, and demonstrate extensive testing.
20	As above, but with multiple extensions, outstanding code decomposition and design, and accompanied by an exceptional report which shows independent research and novel ideas.

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/