CSCI 1933 Project 2

2D Arrays - Chess

Due Date: March 14, 2023 (11:55pm)

1 Introduction

Welcome to the second CSCI 1933 project! All students are expected to understand the rules listed below. While some rules may seem unforgiving, all guidelines are made to help the TAs grade efficiently and fairly. As a result, we will generally not make exceptions. Rules listed in the syllabus also apply but may not be listed here.

IMPORTANT: Please read this document and all other documents that are provided for you thoroughly.

1.1 Submission

Project 2 is due on **Tuesday, March 14, 2023** at **11:55pm CST** on **Canvas**.

Submit a zip or tar file on Canvas containing **all your .java files and your README.** You are allowed to change or modify your submission, so submit early and often, and verify that all your .java files are in the submission. Failure to submit the correct files will result in a score of zero for all missing parts. Late submissions will be penalized in accordance with the syllabus. Make sure to include a README.txt in your submission that contains the following information:

- Group members' names and x500s
- Contributions of each partner (if working with a partner)
- How to compile and run your program
- Any assumptions
- Additional features that you implemented (if applicable)
- Any known bugs or defects in the program
- Any outside sources (aside from course resources) consulted for ideas used in the project, in the format:
  - idea1: source
  - idea2: source
  - idea3: source
- Include the statement: **"I certify that the information contained in this README file is complete and accurate. I have both read and followed the course policies in the 'Academic Integrity - Course Policy' section of the course syllabus."** and type your name(s) underneath.

You may be penalized for a missing or incomplete README.

1.2 Working with a partner

As discussed in lecture, you may work with **one** partner to complete this assignment. If you choose to work as a team, ***please only turn in one copy of your assignment***. Include both of your names and x500s in a comment at the top of each file you submit. In doing so, you are attesting to the fact that both of you have contributed substantially to completion of the project and that both of you understand all code that has been implemented.

1.3 Identification

Include your name and x500 in a comment in all files you submit, even when not working with a partner. Example: // Written by Christopher Dovolis, dovol002

1.4 Questions

Questions related to the project can be discussed with peers in lecture/lab/Discord only in abstract. Such questions might relate to programming in Java, understanding the writeup, or topics covered in lectures and labs. Do not post any code or solutions publicly. Questions that require explaining your solution or posting your code can be asked to TAs in office hours or through help tickets in Discord.

1.5 Grading

Grading will be done by the TAs, so please address grading problems to the TA who graded your project privately.

1.6 Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.
- Use effective and standard indentation.

- Use descriptive names for variables. Use standard Java style for your names: `ClassName`, `functionName`, `variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments. `int a = 5; //Set a to be 5` is not helpful.
- Disorganized and messy files. Poor indentation of braces (`{` and `}`).
- Incoherent variable names. Names such as `m` and `numberOfIndicesToCount` are not useful. The former is too short to be descriptive, while the latter is much too descriptive and redundant.

The programming exercises detailed in the following pages will be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

1.7 Project Overview

Project 2 involves developing a simplified version of the popular board game chess. You will utilize 2D Arrays to represent a chess board, and will write functions that enforce the rules of the game.

Classes you must create (More on this in Sections 2 & 3):

- `Game.java` (SUBMIT)
- `Rook.java` (SUBMIT)
- `Bishop.java` (SUBMIT)
- `Knight.java` (SUBMIT)
- `Queen.java` (SUBMIT)
- `King.java` (SUBMIT)
- README.txt (SUBMIT)

Classes you must edit (More on this in Section 2):

- `Board.java` (SUBMIT)
- `Piece.java` (SUBMIT)

Classes you are given (Do not edit these):

- `Pawn.java`
- `Fen.java`

1.8 Chess

Chess is a centuries-old game of logic and skill originating from India. The game is played on an 8x8 grid between two players. Each player has 16 pieces: 8 Pawns, 2 Rooks, 2 Knights, 2 Bishops, 1 Queen, and 1 King. Traditionally, one player's pieces are white and the other's are black. Each piece has its own set of rules about where it can move, and how it can "capture" the other player's pieces. The goal of the game is to capture your opponent's King.

Chess is a game of many rules, but we have reduced it slightly to a set of rules which we think you can implement. We will discuss the specific rules that must be followed in section 3. Unless a rule is specifically mentioned in this writeup, assume that you need not implement it.

2 Structure

This section outlines the overall structure of the chess app you will be writing.

IMPORTANT: **IntelliJ will be required for this project**, as you will be creating a project file and running JUnit tests on your code. If you need to review how to create a new project in IntelliJ, please refer to the *"Setting up IntelliJ"* section from the Lab 4 document. Also, please read the *testing* document on how to install and run JUnit tests on IntelliJ.

IMPORTANT: Things to keep in mind: **IntelliJ dark mode will make the white pieces appear black and black pieces appear white in the terminal**. Also, **for the sake of testing and grading, white pieces will be required to go upward and black pieces will be required to go downward (in terms of how they appear on screen)**. Dark mode changing the colors of the pieces has caused students in the past to have white going downward and black going upward, so please keep these notes in mind for this project.

2.1 Board class

For this project you will be required to implement a class called `Board.java`. This class will serve as the representation of a chess board. Your board must be represented as a 2-Dimensional array of type `Piece`. Using this structure allows us to keep each piece on the board while representing them with different classes, but more on that in the next section.

We have already given you method signatures for this class, but it will be your job to implement algorithms that achieve their stated goals.You are welcome to create your own helper methods if you want, but you should not modify the existing method signatures as this will interfere with any tests we run as part of grading this project.

As a part of `Board.java`, there are several functions that you will need to implement.

- `public Board()`: Constructor that initializes the 2D `Piece` array representing the board.
- `public Piece getPiece(int row, int col)`: Gets the piece at a location.
- `public void setPiece(int row, int col, Piece piece)`: Sets the piece at a location.
- `public boolean movePiece(int startRow, int startCol, int endRow, int endCol)`: Moves the piece at the start location to the end location, provided it is legal to do so. **Conditions:** There must be a piece at (startRow, startCol) and the piece must legally be able to move to the destination square.

  > **Hint:** Remember, in order to check that a `Piece` can legally move, you can use the `isMoveLegal` function.

- `public boolean isGameOver()`: Checks if the board is in a game over state or not (if the board does not contain a King of each color, then the game is over).
- `public void clear()`: Sets every square in the board equal to null.
- `public String toString()`: Returns a string representation of the board. See the section on Unicode characters for additional details.

In addition to the above methods, you will also implement several methods that will be useful in your piece classes to determine the legality of a move.

- `public boolean verifySourceAndDestination(int startRow, int startCol, int endRow , int endCol, boolean isBlack)`: This function performs verification checks on the source and destination of a move and returns `true` if all of the checks pass.
  Conditions:
  - startRow, startCol, endRow, endCol must be within the bounds of the board.
  - The start position must contain a piece.
  - The color of the starting piece must match the color provided to this function.

  - The destination location must either contain no piece or must contain a piece of the opposite color. (a piece of the opposite color is said to be "captured" when this piece moves to the end location)

- `public boolean verifyAdjacent(int startRow, int startCol, int endRow, int endCol)`: This function verifies that the source and destination of a move are adjacent squares (within 1 square of each other). For example, this is how the King moves.
- `public boolean verifyHorizontal(int startRow, int startCol, int endRow, int endCol)`: This function performs verification checks for horizontal movement and returns `true` if all of the checks pass.
  Conditions:
  - The move must take place in the same row.
  - All spaces on the board between the start and the destination must not contain a piece.
- `public boolean verifyVertical(int startRow, int startCol, int endRow, int endCol)`: Similar to `verifyHorizontal`, but for vertical moves.
- `public boolean verifyDiagonal(int startRow, int startCol, int endRow, int endCol)`: Similar to `verifyHorizontal` and `verifyVertical`, but for diagonal moves.

> **Hint:** The pattern for detecting whether two squares are on the same diagonal can be difficult to spot if you've never seen it before. The pattern is different, but related, for diagonals going one direction vs. diagonals going the other direction. Consider the following two examples. What patterns do you notice between the row number and the column number?
>
> | row | 0 | 1 | 2 | 3 | 4 |
> |---|---|---|---|---|---|
> | col | 4 | 3 | 2 | 1 | 0 |
>
> | row | 3 | 4 | 5 | 6 | 7 |
> |---|---|---|---|---|---|
> | col | 0 | 1 | 2 | 3 | 4 |

> **Note:** While not required, it can often be helpful to use `Math.min`, `Math.max`, and `Math.abs` to take advantage of the inherently symmetrical nature of many of these checks. You are encouraged to use these functions to reduce the amount of code you need to write.

2.2 Piece class

The Piece class is very important for the operation of this project. As you've learned in lecture, Java arrays cannot be defined to have more than one type. This presents a problem for us, as we want to represent 6 different types of piece on the board at once. Enter `Piece.java`. An object of type `Piece` contains an instance variable which is initialized to a Unicode character for one of the 6 chess pieces. Using this char variable, `Piece`'s `isMoveLegal()` will redirect your query to the

proper method for rule-checking. This function is included in the codepack, and you should inspect it to see how you might use Unicode characters and how you may implement pawn promotion as well.

2.2.1 Unicode

Unicode characters are very important for this project, as they give us a very easy was to represent the chess pieces in terminal. Unicode is an expanded library of characters designed to make up for ASCII's lack of support for non-Latin alphabets. On top of supporting many languages, Unicode also has many miscellaneous symbols. Luckily for us this includes both white and black chess pieces! Here are the Unicode "codepoints" for each white and black piece, respectively:

Piece	White	Black
King:	U+2654	U+265A
Queen:	U+2655	U+265B
Rook:	U+2656	U+265C
Bishop:	U+2657	U+265D
Knight:	U+2658	U+265E
Pawn:	U+2659	U+265F

When using Unicode characters in Java, we define a variable of type `char`, and initialize it with the codepoint. For example, to initialize a white pawn we say: `char pawn = '\u2659'`. Your computer's terminal may not support Unicode right out of the box, but IntelliJ should and is easy to configure if not. One thing to note when printing the board to the terminal is that the Unicode chess pieces do not occupy the same spacing as a typical "space" character. Instead, we will use one of the Unicode space characters, `'\u2001'`, which has the same spacing as the pieces. You may also take advantage of special unicode characters for the row and column numbers. Integers 0...9 are coded with `'\uFF10'`...`'\uFF19'`.

Note: There are **two bugs** to look out for with these Unicode characters. One bug is that since Unicode also encodes emojis, **the black pawn occasionally prints as an emoji instead of the plain text character**. This is problematic because the emoji occupies more horizontal space than the plain text pieces. This throws the spacing of our board off by a bit, *but you will not be penalized for that.* Another bug is that because you are required to use IntelliJ, most students uses its "dark mode" feature. **This will make white pieces appear black and black pieces to appear white**, please keep this in mind when displaying your chess board, *but you will not be penalized for this either.*

2.3 Game class

`Game.java` is the class which contains this project's main method. Within this main method you must create an interactive terminal experience for the players to interact with throughout the game.

This class will instantiate the `Board` object and prompt the user for input. A visual representation of the board must be printed at the beginning of each player's turn. The user should enter a starting point and an ending point, but all other work to execute the turn should happen elsewhere. The game should end when one player's King has been "captured".

Note: As a sample of what a turn could look like, consider this position which begins in the final state of the famous "Immortal Game" and, in this case, ends when the white bishop captures the opponents King.

```
Board:
    0  1  2  3  4  5  6  7
 0 |♜|  |♝|♚|  |  |  |♜|
 1 |♟|  |  |♟|♗|♟|♘|♟|
 2 |♞|  |  |  |  |♞|  |  |
 3 |  |♟|  |♘|♟|  |  |♙|
 4 |  |  |  |  |  |  |♙|  |
 5 |  |  |  |♙|  |  |  |  |
 6 |♙|  |♙|  |♔|  |  |  |
 7 |♕|  |  |  |  |  |♝|  |

It is currently white's turn to play.
What is your move? (format: [start row] [start col] [end row] [end col])
1 4 0 3
White has won the game!
```

2.4 Fen class

We have supplied you with a class called `Fen.java` which is a tool for both setting the board to its starting state, and testing your work. It is based on Forsyth-Edwards Notation which is a method for saving the position of every piece on a chess board in 71 characters or less (usually much less). Understanding the way it works isn't critical for the project, but you can read more about it here.

The class contains only one method:

- `public static void load(String fen, Board b)`
  Input: A FEN code in `String` format, and an instance of the `Board` class.

Because this method is static, you will need to reference the entire class when calling it. To initialize starting position, this may look like:

```
Fen.load("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR", myBoard)
```

Note: Don't worry if that string looks a little scary, we have included a handful of useful strings that you can use when testing. You need not know how to make them yourself.

3 Chess Pieces

In addition to creating `Board.java`, you will also be required to implement classes for each of the different types of chess pieces. These classes will all follow a similar structure, with

- A constructor, which takes in parameters `(int row, int col, boolean isBlack)`
- `public boolean isMoveLegal(Board board, int endRow, int endCol)`
  Input: The state of the board, the destination row of the move, and the destination column of the move.
  Output: Whether the piece can move from its current location to the destination provided, given the current board state.

IMPORTANT: Please look at the *rules* document for more information on how to implement certain pieces, especially the pawn with its pawn promotion.

4 Unit Tests

As mentioned at the beginning of this project document, you will be using JUnit tests to test your code. A document titled *testing* explains how to run these tests.

IMPORTANT: Please make sure to to test all of your code as a good portion of this project's grade will depend on your JUnit tests passing or failing.

Note: Please **do not include** the JUnit test file when submitting.

5 Grading

- Individual piece classes: 7 points each
  - `Rook.java`
  - `Knight.java`
  - `Bishop.java`
  - `Queen.java`
  - `King.java`

- Move verification methods: 7 points each
  - `verifySourceAndDestination()`
  - `verifyAdjacent()`
  - `verifyHorizontal()`
  - `verifyVertical()`
  - `verifyDiagonal()`
- `Board` class `toString()` method: 5 points
- Game functionality: 10 points
- Programming style: 15 points