

# Assignment 2

## COMP 250, Winter 2022

Prepared by Prof. Michael Langer and T.A.s

Posted: Fri. Feb. 11 , 2022

Due: Fri. Feb 25 at 23:59 (midnight)

[document last modified: February 16, 2022]

### General instructions

- Search for the keyword **Updated** to find any places where the PDF has been updated.
- T.A. office hours will be posted on mycourses under the “Office Hours” tab. For zoom OH, if there is a problem with the zoom link or it is missing, please email the T.A. and the instructor.
- If you would like a TA to look at your solution code outside of office hours, you may post a *private* question on the discussion board.
- We will provide you with some examples to test your code. See file ExposedTest.java which is part of the starter code. If you pass all these tests, you will get at least **[Updated: Feb. 14] 50/100**. These tests correspond exactly to the exposed tests on Ed. We will also use private tests for the remaining points out of 100.

We strongly encourage you to come up with more creative test cases, and to share these test cases on the discussion board. There is a crucial distinction between sharing test cases and sharing solution code. We encourage the former, whereas the latter is a serious academic offense.

Ed should be used for code submission only. You should write, compile and test your code locally (e.g. in Eclipse or IntelliJ) for development.

- **Policy on Academic Integrity:** See the Course Outline Sec. 7 for the general policies. You are also expected to read the posted checklist PDF that specifies what is allowed and what is not allowed on the assignment(s).
- **Late policy:** Late assignments will be accepted up to two days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Please see the submission instructions at the end of this document for more details.

# Introduction

## Your Mission

Chess has seen a resurgence in popularity in the past years and as such, we're asking you to make a digital chess game. Your task more specifically is to create a program which acts as a sort of "referee" for a simplified variant of chess by making sure that each player makes only valid moves, by tracking the history of the game, and by announcing a winner.

In software engineering, we call such a program an "API" or "application programmer interface" as it exposes some methods which allow other *programs* to interface with it. More formally, an API specification is a set of rules delineating communication standards between two parties (eg. the method descriptions later in this document) and an API implementation is what you implement given this specification (eg. the program that you submit for this assignment). The program you will write will be used to drive an interactive visualizer which we will supply to you and will also be used by our grading program.

Our objective in writing this assignment is to make it fair to students who are unfamiliar with chess, so as many game specific nuances as possible have been pre-implemented for you, leaving just the basic game rules and object oriented programming part for you to do. *Note that advanced chess concepts such as en passant, castling, promotion, stalemate, checkmate, etc. are not considered in this assignment.*

You will be graded using a series of tests (both exposed and private) similar to Assignment 1. You will not be graded on how your code works with the visualizer. It exists solely as a fun debugging tool.

## Summary of the game

Here are the basic rules of our game. Note they differ slightly from standard chess to make things easier.

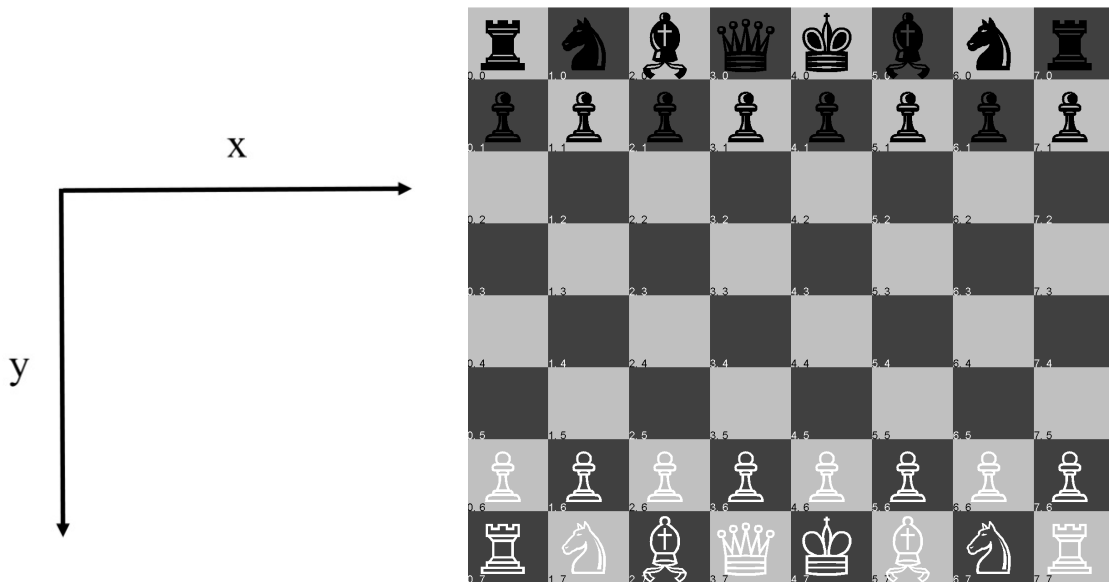
- The game is played on an 8x8 grid of squares where each square is either empty, or contains a "piece". Each piece is colored black or white.
- Two players (black and white) take turns moving pieces of their respective color around on the board. White moves first.
- There are 6 piece "types" (King, Queen, Pawn, Bishop, Knight, Rook). Each piece behaves slightly differently. See **PieceRules.md** for more details.
- If a piece moves to a square occupied by an opponent's piece, the opponent's piece gets "captured" and removed from the board.
- The objective of the game is to capture the opponent's King. If you manage this, you win. Note this is intentionally different to standard chess rules, whereby the game ends in a "checkmate" state. We do not model checkmate in this assignment.

## Features

Listed below are features of chess rules that are required to be implemented for this assignment. For those of you who are already familiar with chess, please do not implement features/rules other than the ones we require, as this will cause a problem with the grader.

## Board indexing

The squares on the board array are indexed using  $[x][y]$  where  $(x,y) \equiv (\text{column}, \text{row})$ . The origin  $(0,0)$  is at the upper left, and  $x$  is to the right and  $y$  is downward. Note that this is different from how 2D arrays were visualized in lecture 5 (slide 31), which used  $(\text{row}, \text{column})$  indexing instead.



## Turn Order

White moves first, then black, then white, etc. You must not allow a player to make a move out of turn.

## Piece Movement

Each player (black and white) has 16 pieces with the following names, and number of each is shown in brackets: pawn (8), rook (2), knight (2), bishop (2), queen (1), king (1). Each piece is restricted to moving in a particular pattern. There are many web resources that explain how each type of piece can move, for example:

- [chess.com](http://chess.com)
- [chesscentral.com](http://chesscentral.com),

One of your main tasks will be to write the code that allows each piece to move according to the rules of the game. These rules are provided to you later in this PDF with code samples. See also the file `PieceRules.md`.

## Capturing

If a piece moves to a square where an opponent's piece is located, the opponent's piece is "captured", meaning that it is taken out of the game.

## Check and Win Conditions

A player is said to be "in check" if their king is "attacked" by an enemy piece, i.e., it is at risk of capture by any of the opponent's pieces in the next move.<sup>1</sup> You will be responsible for writing a method which tells you if a player is in check.

As previously described, to win the game you must capture the opponents' king. This means that in our version of chess, a player is allowed to move into check and allow their king to be captured, and if a player is in check, they are not required to resolve it, i.e., they are allowed to make any legal move and allow their king to be captured by the opponent. This differs from standard chess, where the player must, if possible, immediately resolve a check either by moving the king, blocking the attack, or capturing the attacking piece.

Note that we are not modelling or testing any cases where a player has no valid moves (e.g. if the black king is in the far corner, surrounded by black pawns) since such cases are very contrived.

## Pawn Promotion

In standard chess, when a pawn reaches the other end of the board, they are forced to "promote", or turn into, any piece except the king or pawn. Since we are not modelling this rule, pawns that reach the other end of the board should *remain* as pawns in your implementation.

## Game History

Most digital chess games have a feature where every move that is made is stored in a list, letting players "rewind" through the game after it is finished. Similarly, our game will track every move that is made and at the end of the game, it will print every move that was made.

Note that in our version of chess, it is possible for a single move to put both kings into check simultaneously (e.g., if the white queen is blocking a check from the black queen, but then moves out of the way to deliver its own check to the black king). We will not be testing such cases.

## Your Task

You will need to create and implement several classes. You will also need to complete the implementation of several classes that are provided to you but are only partially implemented. Note that the starter code that is provided to you will not compile until you have created all of the following Piece classes and you have provided at least some implementation of their methods – even if they are very basic and incorrect.







---

<sup>1</sup>In regular chess, there is also an important game state called checkmate, but we do not model this.

[updated Feb.15 ] We also recommend you begin this assignment by implementing the Piece classes first, then move to the Game class.

## Piece

First are the classes that extend the **Piece** class. The symbol for this piece will be shown on the board, and so you will need to learn to recognize it.

- **King**  (a skeleton implementation for king has been provided for you to get started)
- **Queen** 
- **Bishop**  (a skeleton implementation for bishop has been provided for you to get started)
- **Rook** 
- **Knight** 
- **Pawn** 

For the above classes, you need to implement the following methods:

- **constructor**: see code provided in the Bishop piece, for example
- **canMove(int destX, int destY)**: This method takes as input the coordinates of a destination square and verifies if this piece is allowed to move to this square (as per piece movement rules described earlier); it returns a boolean. Note that you only need to consider if the move would be allowed on an otherwise empty chessboard, namely it should *not* consider the interaction of this pieces with any other pieces on the board (except for the Pawn class, which sometime can move diagonally, as described later in this pdf); the question of whether other pieces are obstacles to the move are handled by the Game class's canMove() method described later.  
  
[Updated: Feb. 14] The Piece.canMove() method should not test the case where (destX, destY) is the same square that the piece is currently on. That test will be handled by the Game.canMove() method.
- **getSymbol()**: Snippets of code for this method are given to you in PieceRules.md and later in this document.

Note that the Piece abstract class itself has several implemented methods that are given you. You should not touch this abstract class. Instead, you should implement its two abstract methods canMove() and getSymbol() for each of the above classes which extend Piece. [Updated: Feb. 15] We have removed the Piece.isCaptured() method and the captured field since they are unnecessary

## Side

In the starter code we provide you with a file called `Side.java`. Within it is an “enum” which can only be one of two values, either `BLACK` or `WHITE`. We use this enum throughout the project to track the colors of pieces and to track whose turn it is. For example, to set `x` to `WHITE` you can do **`Side x = Side.WHITE;`**. Note that an enum is *not* a string, so to get the string “WHITE” back from `x` you must do **`x.toString();`**.

[updated Feb. 13] The enum also has a static method **`Side.negate(Side s)`** for flipping colors. Passing it `WHITE` will return `BLACK` and passing it `BLACK` will return `WHITE`.

DO NOT modify this file.

## Game

You also need to implement some of the methods from the **Game** class. This class has the following member variables:

- **Board b**: Keeps track of the state of the game (positions of pieces, captured status etc.).
- **Stack< String> moveHistory**: Keeps track of every move made during a game.
- **Side currentTurn**: Keeps track of which player’s turn it is currently.

You are required to implement the following:

- **Game()**: This constructor creates a new game. Specifically, it should:
  - Create a new **Board** object
  - Assign the correct starting value to **currentTurn**.
- **canMove(int x, int y, int destX, int destY, Side s)**: This method returns true if the player of color `s` can move the piece at coordinates `(x,y)` can move to coordinates `(destX, destY)` on the board in its current state. This means that here you *do* need to consider this piece’s interaction with other pieces on the board. Conditions for this method to return false are given in the code.
- **move(int x, int y, int destX, int destY)** : This method takes in coordinates of a piece at `(x,y)` and a destination `(destX,destY)` and does the following: if it is a valid move then it tracks the game history (see below), moves the piece, updates whose turn it is, and returns true; otherwise it returns false.
  - Move validity is determined by the **canMove(...)** above, so you will need to implement **canMove(...)** before this. If the move is invalid, return false.
  - To actually move a piece, you must invoke the piece’s **move()** method. Thus, the **Game**’s **move()** method must call the **Piece**’s **move()** method.
  - Make sure you update the current turn only if a move was made.
  - To track game history, we require you to call our methods: [updated Feb. 15:] **appendMoveToHistory()** must be called first.

- \* **appendMoveToHistory(int x, int y, int destX, int destY, Side s):** Records to game history that piece belonging to Side *s* at (x,y) moved to (destX,destY).
- \* **appendCheckToHistory(Side s):** Records to game history that the king of Side *s* is in check. As described previously, you can assume that a single move will not result in checks to both sides. [Updated: Feb. 15] However, the method `appendCheckToHistory` *should* be called if a player accidentally puts themselves into check, as well as if a player puts their opponent into check.
- \* **appendWinToHistory(Side s):** Records to game history that Side *s* has won the game. Note that unlike regular chess, in our simplified assignment, a side wins the game when it *captures* the opposing king. [Updated Feb. 14]: the starter code didn't originally specify this requirement in the comments. It will be added.
  - [updated Feb. 15:] If the source and destination squares are the same, you should return false.
  - If a piece was actually moved, this method returns true, otherwise false.
- **isInCheck(Side s):** Returns true if the king of side *s* is attacked by any of the opponent's pieces, i.e., if in the current board state, any of the opponents pieces can move to where the king is. Otherwise, it returns false. Note that this method is only used to warn the player when they are in check. You can use the GUI to test if this is working.
- [updated Feb. 15:] Don't worry about what happens after a King is captured so long as the history move/check/win are updated properly.

## Board and GameApp

Two classes are fully implemented for you. The **Board** class encodes the current state of the game and handles your interactions with it. In particular, it carries out the moves, resets the game, etc. Essentially, the state is a 2D array of type **Piece**. Note that due to polymorphism, this array can hold objects of any subclass of **Piece**. Although you do not need to touch this class, you should familiarize yourself with how it works.

The **GameApp** class creates a new Game and launches the graphical user interface that shows the board state. This class essentially uses all other classes mentioned above to render a visual representation of the current state of the game. You do not need to look into how this works, but should know how to use it as described below. You may use it to quickly check if your implementation is working as expected.

To launch this, run the main method in **GameApp** class. A blue square represents a selected piece. Green squares represent places where the selected piece can move to, according to your implementation.

- You can Left click a piece to select it. If your 'turn tracking' logic is implemented correctly, you will only be able to move white first, then black and so on.
- If your `canMove()` and `move()` methods are implemented correctly, the green squares that appear after selecting a piece will show you all legal moves for that piece per our rules of chess.

## Piece Rules

We provide to you a list of Java expressions that depend on:

- `this.x` : the x coordinate of the piece to move
- `this.y` : the y coordinate of the piece to move
- `destX` : the x-coordinate of the square to move to
- `destY` : the y-coordinate of the square to move to

In order to implement your own logic, you can also use this variable in your methods:

- Board `b` : the board on which the piece is moving

You are intended to use these expressions to implement `canMove()` for the Piece class. You are intended to use the symbols in the `getSymbol()` method.

**King**  : 

```
(Math.abs(this.x - destX) <= 1 && Math.abs(this.y - destY) <= 1)
```

**Queen**  : 

```
(Math.abs(this.x - destX) == Math.abs(this.y - destY)) || (this.x ==  
destX || this.y == destY)
```

**Rook**  : 

```
(this.x == destX || this.y == destY)
```

**Bishop**  : 

```
(Math.abs(this.x - destX) == Math.abs(this.y - destY))
```

**Knight**  : 

Note that the knight can “hop over” pieces, both friendly and enemy. The knight is the open piece that can do this. You are responsible for implementing this behaviour in `canMove()`.

```
(Math.abs(this.x - destX) == 2 && Math.abs(this.y - destY) == 1) || (  
Math.abs(this.x - destX) == 1 && Math.abs(this.y - destY) == 2)
```





## Pawn :

The pawn has a few special and unusual rules for how it can move. First, the pawn normally can advance forward one square per turn. However, if the pawn has not yet moved (it is on its starting square) then it is allowed to move forward either one or two squares. In the case of a two square move, it is not allowed to "hop" over other pieces; recall only the knight can hop over pieces.

Another special rule is that if there exists a piece directly in front of the pawn, then the pawn cannot move forward. In particular, if it is an opposing player's piece, then the pawn cannot capture that piece. So how does the pawn capture other pieces? There is only one way: if an enemy piece exists one square diagonally forward from the pawn (one square forward and then one square left or right) then the pawn may capture it and move to that diagonal square. We emphasize that the pawn must always move forward, either straight forward to an empty square, or diagonally to capture an enemy piece.

You are responsible for implementing all the above described behaviours for all pieces.

### Tips & Tricks

- You can use the ternary expression to pick a symbol based on the side of the piece. For eg.  
`this.getSide() == Side.BLACK ?  :  ;`
- Use your IDE's code navigation features to quickly jump around in the codebase to look at various function definitions and documentation. For eg., you can Ctrl+click on a method in windows IntelliJ to go to its definition.
- You can split your coding screen to open two different source code files side-by-side, saving you the trouble of having to constantly switch between tabs.

## Submission

Please follow the instructions on Ed Lessons Assignment 1.

- Ed does not want you to have a package name. Therefore you should remove the package name before uploading to Ed.
- You may submit multiple times. Your assignment will be graded using your most recent submission.
- If you submit code that does not compile, you will automatically receive a grade of 0. Since we grade only your latest submission, you must ensure that your latest submission compiles!
- The deadline is midnight Fri. Feb. 25. On Ed, this deadline is coded as 12:00 AM on Sat. Feb. 26. Similarly, on Ed, the two day window for late submission ends at 12:00 AM on Mon. Feb 28.

Good luck and happy coding!