Chapter 7 Labs

Getting Started

Follow these instructions from the Chapter 4 Labs to get started.

Chess

This exercise involves the well-known strategy game of chess. If you aren't familiar with the basic rules, refer to this page (Note that this lab doesn't require knowledge of some more advanced rules like *en passant*).

Overview

Before beginning, there are some important details regarding the game software implementation you may need to know. Each game has its own `GameLoop` which coordinates the basic turn-by-turn operation of the game. The loop maintains several components including two `Player`s, one `Board`, and one `GameWindow`. The `GameWindow` contains some code for the graphical portion of the codebase; its implementation is unimportant and will be disregarded. `Player` instances represent either human or computer players. Each `Player` has a `makeMove()` method that takes both a `Board` and the last opponent `Move` and returns its `Move`.

The `Board` is a more complex entity that holds the entire state of the game at any given moment. This includes a list of `Piece`s. Each piece has a (row, col) position on the board. Like a 2D array, the squares are indexed from the top left and start at (0, 0). Additionally, each piece has internal logic that describes how it moves as well as a `Color` (white or black). (Side note: in the code, colors are represented using a new kind of Java entity called an enum. If you're curious, you can read more about enums here) The `Board` class also has some useful methods that can detect mate and remove invalid moves (e.g. moves that keep/put the current player in check) among other things.

To see all of this working in concert, run the `Main.java` file. This starts up a human-human player game (we'll learn later how to change the players). You should be able to see a properly setup chess board; however, most of the pieces can't move. We'll fix that in the first exercise.

Network Operation

This section is irrelevant to the tasks of the lab so feel free to skip it. In addition to local matches, the software can also have games over the network. In each networked match (and most networked communication infrastructure), there is a client (always white) and a server (always black). Consequently, both a client and a server computer are required to start a networked match. To start a game client, run the `NetworkClientMain.java` program. This will open a window showing a list of all available games (broadcasted by servers) to connect to. When the desired game appears, select and press "Connect". To start a game server, run the `NetworkServerMain.java` program. This will prompt you for a name identifying this match to all possible clients. After you've inputted a name, the game will start when a client connects.

Basic Movement

Your first task is to program the movement of each of the pieces. All pieces must extend the `Piece` class. This abstract class has a single abstract method, `getValidMoves()` that returns a list of valid moves given the current `Board`. This method returns a `List<Move>` where `Move`s are merely simple objects that have a start and end location. To help you out, I've written the basic skeleton for each of the piece subclasses and completely implemented the `Pawn` class. Despite their apparent simplicity, the movement logic for pawns is rather complicated. Make sure to carefully study its `getValidMoves()` method and try to understand how it all works.

There are also two static utility methods in the `Piece` class that may (hint) help with your implementations. However, I caution you not to use them until you fully understand their purpose.

Please fill out the `getValidMoves()` method for the `Knight`, `Bishop`, `Rook`, `Queen`, and `King` classes. When you're finished, you should be able to spin up a new game and play.

Mirror Player

Now that you have a working game framework, you're going to make a simple computer player. To do this, make a new class `MirrorPlayer` that extends `Player`. The intent of this player is to the opponent's plays move-by-move. For example, if the human player moves its queen's pawn forward one square, the mirror player should do the same with its queen's pawn. You may safely assume that the `MirrorPlayer` will always be black. Additionally, the player doesn't need to check that the move it makes is valid so long as it is an exact mirror of white's last move (this may cause an error popup; don't worry about it).

To switch out one of the human players for your `MirrorPlayer`, replace the black player in the `GameLoop` constructor in `Main.java`.

Promotion

In addition to the normal rules, chess has some special rules that only apply in specific situations. In this exercise, we will explore one of them: promotion. Basically, when a pawn reaches the other side of the board, it can be promoted to another piece of the same color (e.g. a queen) other than a pawn.

Your task is to implement promotion in our chess program. To make things easier, pawns will automatically be promoted to a queen in lieu of having players choose the desired promotion. To accomplish this, two methods will need to be modified: `Board#apply()` and `Board#unapply()`. The `apply()` method accepts a single `Move` as a parameter and updates the game state to reflect that move. Correspondingly, the `unapply()` method does the reverse of this (i.e. it undoes a move). To help you understand what these methods currently do (and how you should go about modifying them), here is a brief description of what `apply()` does:

1. Gets the piece at the move start location
2. Gets the piece at the move end location
3. Add the piece to a list of captured pieces
4. Set the location of the start piece to the end location

`unapply()` uses similar logic albeit in reverse. It's very important for the operation of the game that both the `apply()` **and** the `unapply()` methods are modified properly.

Castling

Castling is another special rule in chess. If you are unfamiliar with it or need a refresher, use the link in the first section and read about it.

Your task is to implement castling in our program. As with promotion, we will also be modifying the rule slightly to fit our purposes. In this case, the king may castle if the king and/or the corresponding rook have moved (so long as they are still in the proper spot). The other subtle rules of castling still apply though; for example, the king may not castle through nor into check. One more thing to note is that castling is shown as one of the king's moves (not the rooks'). For example, the two outside moves of the white king correspond to queenside and kingside castling:

This is by far the hardest task for this lab; take your time and try not to get too frustrated. Good luck!