# Kung Fu Chess

Created by Sam Youkeles, Etai Kurtzman, Will Randall, and Ian Hackman

**Project Overview:**

Kung Fu Chess plays like a normal game of online chess – except for the fact that there is no turn order. Instead of playing one move after another, two players join the game and can immediately move any number of pieces at any time. We have implemented a number of features such as piece cooldowns (pieces become inactive for a period of time after they have been moved), pretty graphics to display the game state for each player, a countdown to start the game, and more; each contributes in creating a pleasant user experience.

Our client-server model connects the two players to a single server that hosts the game. Each player sends move requests to the server, the server updates the shared game board state accordingly, and players continuously listen for changes to the board that are broadcasted by the server. Because each player operates within a separate thread, they will attempt to update the board concurrently, meaning that the board must be protected from updates that could potentially happen at the same time. By using appropriate locks and message passing, we have implemented code synchronization to prevent data races and deadlocks that might arise from such concurrent programming.

**Minimum Deliverable:** At minimum, we planned to make a 2-player implementation of Kung Fu Chess that can run with two users on the same server. We believed that this was quite feasible given what we had learned in our concurrent programming class up to that point.

**Maximum Deliverable:** Our end goal was to have a working 4-player chess implementation that can be played across computers on any server. We also wanted to provide more user options such as different cooldown times, additional rules such as en passant, and more chess variants such as "duck chess" and "chess 960". This would be a challenge, but we were hoping that pre-existing python libraries would lighten this load and make it possible to accomplish within the timespan of the project.

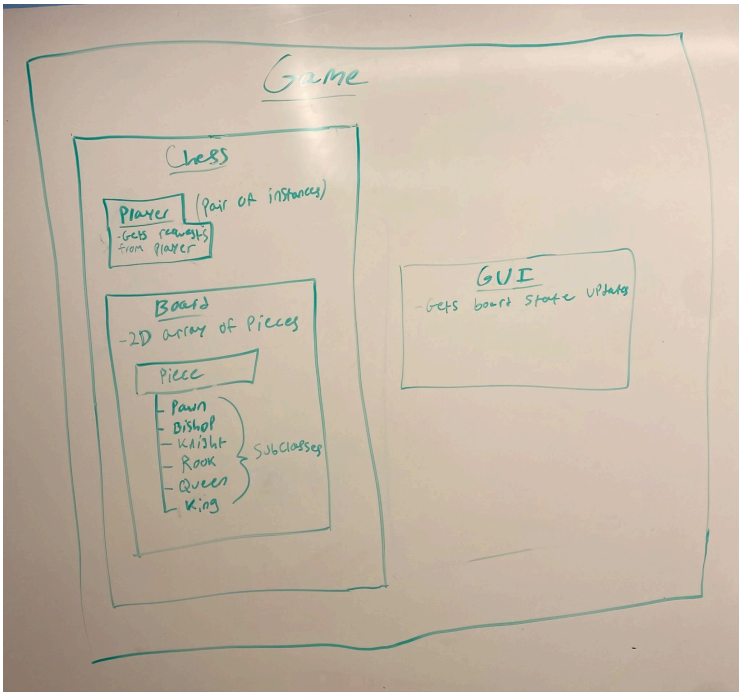
**Outcome Analysis:**

We were successful in achieving our minimum deliverable. We found that the concepts we had learned in our concurrent programming class were very applicable, such as the client-server model and the use of mutexes. We determined that we would program this project entirely in Python, and make use of the Socket library to establish client-server connection. We made use of message-passing to pass information between players and the server, and we developed our message protocol as we iteratively improved our design.

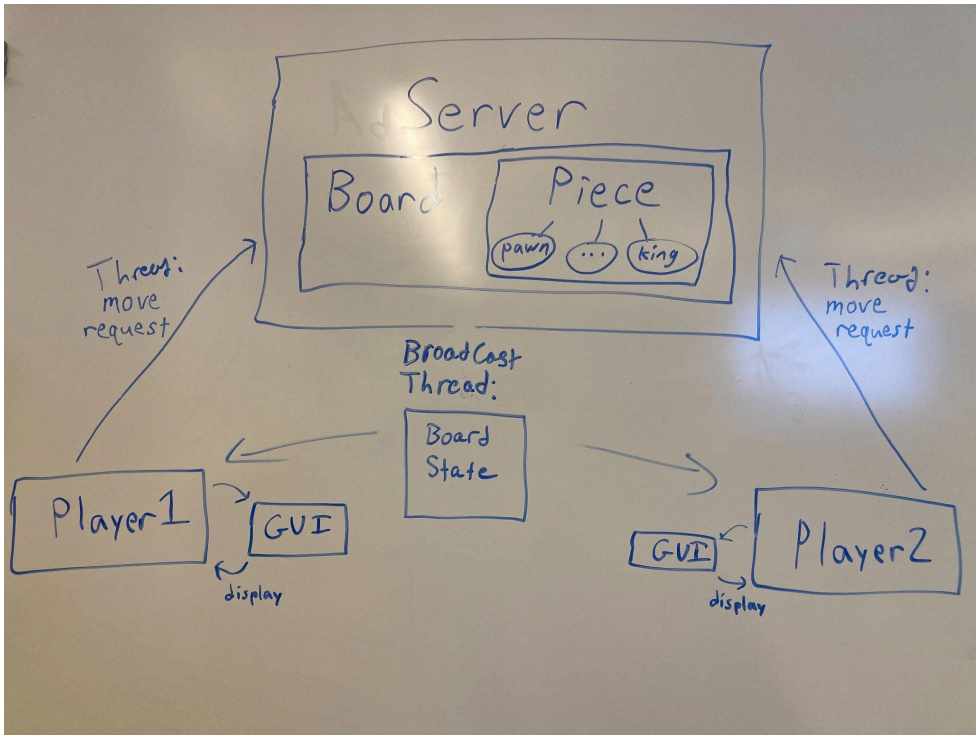
The only aspect of the maximum deliverable that we were able to attain was the piece cooldowns. This required more concurrent programming than we had expected, because it required us to create a new thread every time a piece was on cooldown (to allow for parallel timers). We were hoping to implement variants of the standard chess game, but determined that it was more important to hunt down and fix bugs before expanding the scope of our project. That being said, we took a little bit of additional time to make sure the game had a pleasant user experience. In the end, our game was completely playable and met all of our basic requirements.

**Iterative Design Process:**

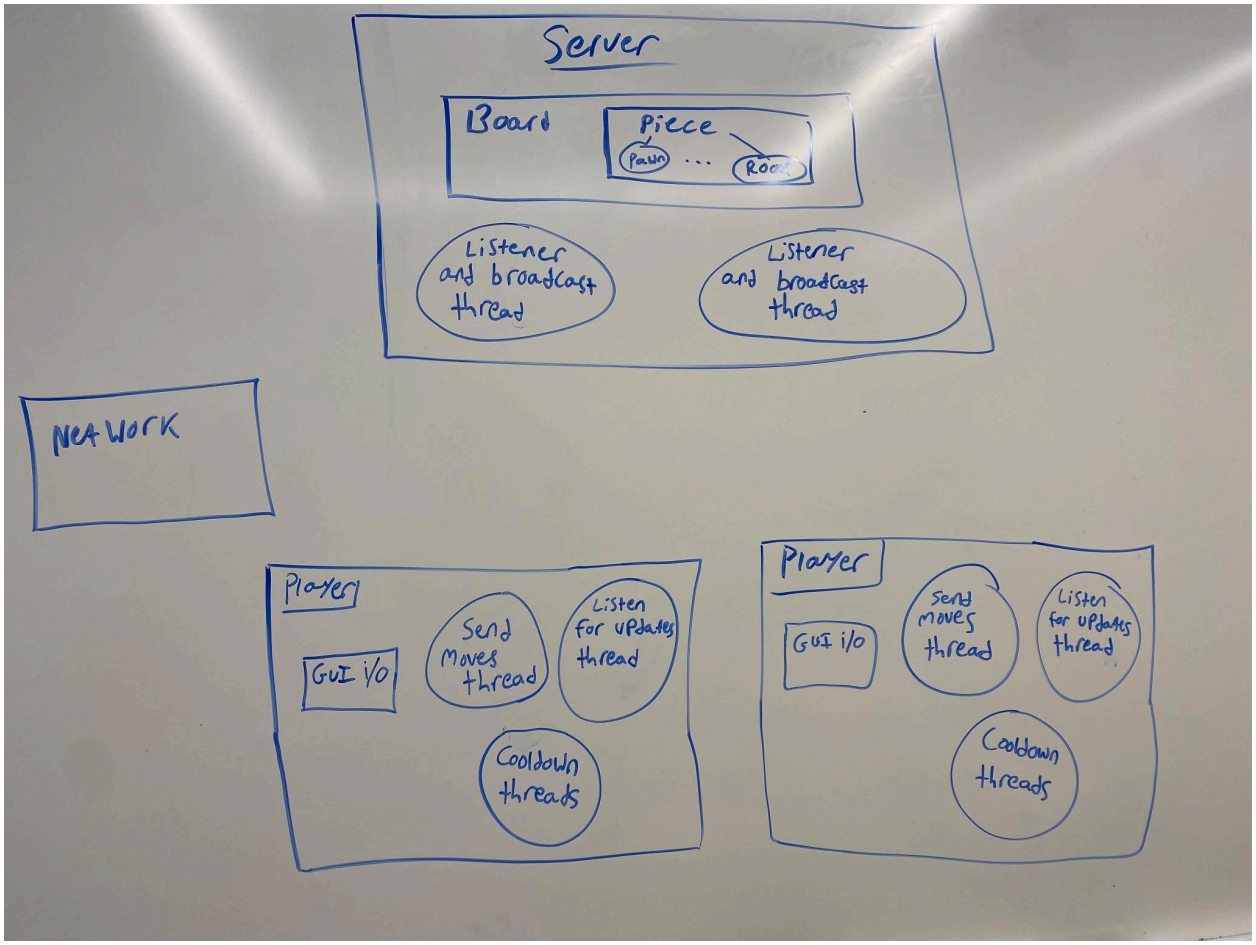
Our initial design included a solid understanding of how to represent the board and the pieces, ensuring that we can enforce the rules of the game and keep track of the current game state. However, our initial design did not include a solid understanding of a client server model, and no understanding of how to leverage network libraries.




Our refined design included the basis for our client server model. It showed some understanding of which threads would be running in each module, but did not have the clarity that we eventually ended up with.



The biggest change between our previous design and our final design is the addition of cooldown threads. These threads weren't something we considered at first, and we sort of assumed that animating cooldown would be relatively easy. However, we quickly realized that continuously updating the GUI is not efficient, and we needed a more clever way of animating cooldowns. Thus, we added threads which begin each time a piece is moved, animate a red version of the piece over the square for the given cooldown time, then reanimate the board afterwards. Our server also did not end up having one broadcast thread, but rather the listener threads are also each responsible for broadcasting.



**Class overview** (**Bold** indicates a class, *italics* indicates a thread)**:**

- **Server**
    - Responsible for maintaining and updating the current state of the board.
    - **Board**
        - A 2D array of pieces, with an interface for moving those pieces in a manner that adheres to the rules of chess.
        - **Piece**
            - Responsible for knowing how pieces move, with a subclass for each type of piece. Also knows how long it takes for each piece to cooldown.
    - *Listener and broadcast threads (2 instances) (aka client_loop)*
        - Each thread listens for move requests from one player, updates the shared board, and then broadcasts the updated board to both players.
- **Network**
    - Establishes a TCP socket connection between the players and the server, and provides an interface to send messages between them.
- **Player** (2 instances)
    - GUI I/O (neither a class nor a thread, just a subcomponent)
        - Gets player input and updates GUI output (relies on external Pygame library).
    - *Send moves thread*
        - Gets moves from the player via GUI input, and sends the move request to the server via the network interface.
    - *Listen for updates thread*
        - Receives updated board state from the server.
    - *Cooldown threads*
        - A new thread is started every time a piece moves. It is responsible for displaying a moved piece as red until the cooldown time for that piece ends.

**Design Reflection**

*Best Decision Made:*

The best decision that was made for the project was the design of the client-server structure of our program. The structure of our program works as follows:

- The player tries to move a piece on the visual layout of the board. The player takes the coordinates of the click and sends it to the server.
- The server has the master representation of the board for all players. The server then tries to make the move and if it is successful, it updates the state of the board.
- The server then sends an encoded string representing the layout of the updated board to both players who decode it and use their GUI module to print the board.

This design choice is advantageous because it means that all of the computations are only done by the server and not the players. The players do not have access to the board and are only in charge of input and output. Thus, the board is abstracted from the players. In addition, having only one master representation of the board makes it easier to handle multiplayer and message-passing, as the server listens for messages from both players and sends messages if there is a successful move. Because of our program's modular design, it was easy to isolate bugs and implement new features.

*What we would do differently:*

In the future we would, when initially designing our program, consider the multiplayer framework for our design more heavily. When implementing the program we first implemented a working version of the game on one device, and then for 2 players using the same server. This worked as it allowed for us to focus on the mechanics and playability of the game. However, it is going to be difficult to, in the future, implement a version of the game that supports up to 4 players. For example, the board will need to be different, there will need to be more than 2 colors, the amount and types of messages will need to change, etc. Since the use of 2 players is so ingrained into the code, checking if pieces are white/black will be difficult to change in support of 4 players. Finally, the way the server works now is that a server is created and the game only starts if 2 players join. It does not support having multiple servers or allowing multiple games to happen at once for a more expansive multiplayer experience. Therefore, a lot of restructuring will need to be done to support those changes.

**Division of Labor**

Most of the work for this project, particularly in the beginning phases, was done all together. While this resulted in code being produced at a slower rate, it had the upside that the code was higher quality with less bugs cropping up, and it also allowed us all to understand many aspects of the project well, which resulted in higher quality design decisions down the line.

Once we had a more operable version of the project, we were able to divide responsibilities and independently add extra features on top of our foundation. Examples of work done independently and then later integrated include cooldown functionality, cooldown animation, start screen, end conditions and end screen, and bug fixes with respect to promotion and move legality.

Overall, we are satisfied with our workflow and think it resulted in good progress throughout the course of the project, but in the future we would attempt to, earlier on in the process, delineate independent modules with very clear interfaces that could be developed independently.

**Bug Report**

Our most difficult bug to find occurred when trying to make our pieces on cooldown display differently from other pieces. This posed a bit of a design challenge, since the client was solely responsible for displaying the board, and did not have access to any cooldown information by default. That said, we found a way to send the relevant information from the server, and got to work on displaying the pieces on the client side.

Our bug was multifaceted; at first the pieces would turn red as we expected, but then would disappear after their cooldown time. We were puzzled, but had a few candidates as to what may have been causing the issue. First, we thought that our function in charge of drawing pieces was somehow not recognizing the piece in cooldown after the timer. We tested this by simply adding a print statement so that we could see what the function thought it was doing. Everything appeared fine there. Next, we thought that the thread that was supposed to redraw the piece once its cooldown had ended ran into some trouble when obtaining the lock that granted exclusive access to the canvas. We verified that this was not the issue by making sure the functions called after the lock was obtained were actually called. This again raised no issue.

It took perhaps two hours for us to finally find the issue - it had turned out that there was an error in the order of operations in which we drew the pieces. The piece indeed was being drawn correctly, but then being drawn over by another function. We fixed this bug by redrawing in the correct order, but this was not the end of our troubles.

We also had an issue that the cooldown did not appear to be working properly on the black player's piece display. We kept in the print statements from the previous bug, but everything seemed to be printing as expected. We also found that sometimes when a black piece moved, it would display a white piece as red. This confused us very much at first, but after another 45 minutes or so, we realized that we weren't adjusting the coordinates to the flipped display; it wasn't that the wrong player's piece was being highlighted, it was that the coordinate was reflected diagonally. We fixed this by doing a simple change of coordinates on the tuple that determined the index of the cooling down piece.

In retrospect, we should have tested more frequently before running our code. A lot of these issues could have been caught if we did small tests as we wrote. Additionally, we should have been more exhaustive while finding our bugs. We tended to focus too closely on one possible solution instead of taking a holistic approach, which cost us a lot of time.

## Overview of Code

**server.py:** The computer that is running the server will execute this file. This file creates the game board that will be shared between players and runs two threads that listen for player moves.
**board.py:** The server will contain one board object that keeps track of the state of the game. When the server processes player moves, it will appeal to functions in this file to determine whether the move was valid or not. The board contains objects of the piece class to represent the individual pieces on the board.
**piece.py:** This file defines the piece class, which is a superclass to represent each piece on the board. This class contains each piece's cooldown-related information. Each of the following files is a subclass of the piece class, and includes piece-specific functionality related to the validity of that piece's movements.

- **bishop.py**
- **king.py**
- **knight.py**
- **pawn.py**
- **queen.py**
- **rook.py**

**player.py:** Each computer that a player is using will execute this file. This file handles the player thread that accepts user input, the loop that listens for board updates from the server, and contains draw functions that utilize Pygame functionality to display the GUI for each player.
**network.py:** This file provides functionality to create a socket network that a player will use to communicate with the server. Each player object will have an instance of the network class.
**imgs:** folder containing the PNG images for all white and black pieces as well as the cooldown images for each type of piece.

## Instructions for Use

Dependencies:

- Python **3.10** or later (check using 'python --version')
- Pygame (install using 'pip install pygame')

1. cd into the project directory; NOT imgs or pieces

2. Start the server:
    a. In a terminal enter the following command:
    python3 server.py

```
PS C:\Users\16109\OneDrive\Desktop\CS Stuff\KungFu 22\KFC-main> python3 server.py
```

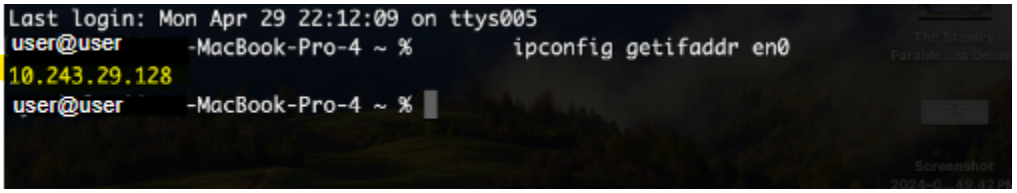
    b. Enter your computer's IPV4 address after the prompt "`Enter your IPv4:`"

```
PS C:\Users\16109\OneDrive\Desktop\CS Stuff\KungFu 22\KFC-main> python3 server.py
pygame 2.5.2 (SDL 2.28.3, Python 3.9.13)
Hello from the pygame community. https://www.pygame.org/contribute.html
Enter your IPv4: 10.243.74.105
```

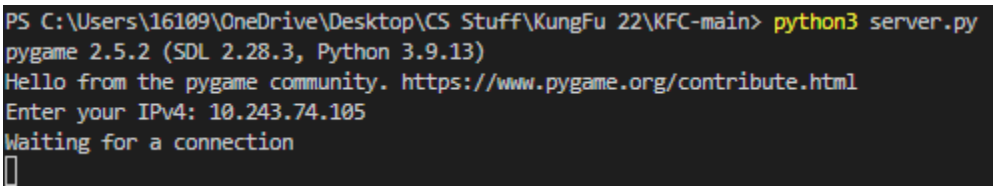
        i. To get your computer's IPV4 address:
            1. Windows: open the command prompt and enter "ipconfig".

```
C:\Users\16109>ipconfig

Windows IP Configuration


Wireless LAN adapter Local Area Connection* 3:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 4:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . : hsd1.ma.comcast.net
   IPv6 Address. . . . . . . . . . . : 2601:19c:417f:b710::9da1
   IPv6 Address. . . . . . . . . . . : 2601:19c:417f:b710:4ded:3820:fc6b:c717
   Temporary IPv6 Address. . . . . . : 2601:19c:417f:b710:20b6:977b:51eb:c624
   Link-local IPv6 Address . . . . . : fe80::df16:2925:fb1:1577%16
   IPv4 Address. . . . . . . . . . . : 10.0.0.253
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : fe80::16c0:3eff:fe4e:a76d%16
                                       10.0.0.1

Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
```

2. Mac: open the terminal and enter “ipconfig getifaddr en0”

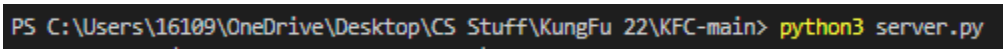



   c. After the IPV4 address is entered the terminal will print the following: “`Waiting for a connection`“

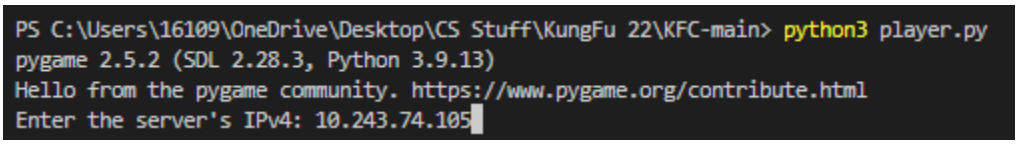



   d. Nothing more needs to be done with the server.

3. For each player to join:
   a. In a terminal enter the following command:
   python3 player.py




   b. Enter the SAME IPV4 address as the server after the prompt: “`Enter the server’s IPv4:`“




   c. The terminal will wait for the other player to connect before displaying the chess window.
   d. Note: Players should be on separate devices, but player and server can be run on the same device. All devices should be on the same network.
   e. Note: Both players should have all code locally

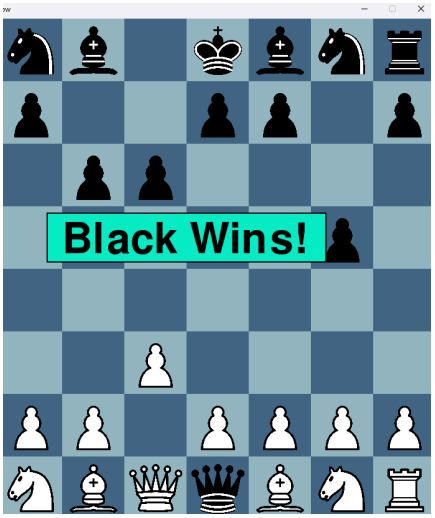
4. To start the game:
   a. Once both players have joined the server the window displaying the chess board will appear for both players.


b. To start the game, each player has to click “Start”.
   i. Once the first player clicks start, the game will wait for the 2nd player to also click start before starting the game.


   ii. Once the 2nd player has joined a countdown will begin! After the screen displays “Go!” both players can start making moves.

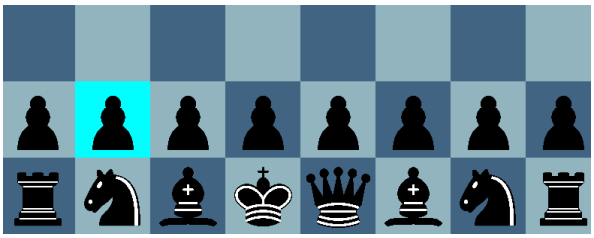

5. Rules
    a. No turn order!
        i. Both players can move whenever they want, as often as they want. They do not need to wait for the other player to make a move.
    b. Piece cooldowns
        i. After a piece has been moved successfully the piece will turn red.
            1. The piece is now in cooldown, meaning it cannot be moved again until after 3 seconds have passed and the piece is no longer red.


    c. Capturing the King wins the game.
        i. There is no checking. The game only ends once one of the player's Kings are captured.

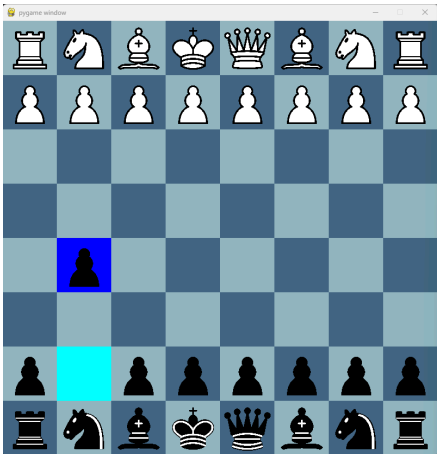



d. No en passant
    i. Since there is no turn order en passant doesn't apply.
    ii. There **IS** castling and pawn promotion

6. Controls: How to move pieces
    a. To move a piece: Click and Drag
        i. Click on your color piece. If the click is valid then the square behind the piece should turn light blue.



        ii. Drag the piece to an empty square or a square with an opposing piece according to the rules of chess.
            1. Here is a link to the rules of chess: [How to Play Chess: Learn the Rules & 7 Steps To Get You Started - Chess.com]
            2. If the move is valid, the piece will move to that square and the square behind the piece will turn dark blue.



7. Starting a new game
    a. Once a player has captured a king the game ends and the server and players will be disconnected. To start another game, start again from step 1
8. Enjoy!

**Credits:**

- We watched this tutorial to get the basics of networking/sockets:
    - https://youtube.com/playlist?list=PLzMcBGfZo4-kR7Rh-7JCVDN8lm3Utumvq&si=sTimad6wr_wNdvEC
- Our Chess Piece Images came from wikepedia:
    - https://commons.wikimedia.org/wiki/Category:PNG_chess_pieces/Standard_transparent