Documentation

A) Printing Board:

When managing the printing of the board representation, I faced several challenges and implemented solutions to overcome them. Here is a detailed account of the issues I encountered and how I resolved them:

I constructed loops to manage the printing of the board representation, ensuring that they could dynamically adjust to different board dimensions. By using nested loops, with the outer loop iterating over rows and the inner loop iterating over columns, I ensured that the entire board would be printed regardless of its size.

To create a visually pleasing representation, I had to address formatting and alignment. This involved aligning each element within its corresponding cell on the board. I used formatting techniques such as padding and spacing options provided by the programming language or library to achieve consistent spacing between elements, resulting in a neatly aligned board representation.

Handling different types of elements on the board was another challenge. The board could contain empty spaces, player tokens, or game pieces. To accurately represent each type, I used conditional statements within the loops to determine the type of each element and printed the appropriate representation accordingly. This approach allowed me to differentiate between different elements on the board and display them correctly.

I also had to consider the limitations of the console window. When the board size exceeded the dimensions of the console window, it could lead to an incomplete or distorted representation. To address this, I implemented logic to break the board into multiple sections and print each section on a separate line. This ensured that the entire board could be displayed within the console window's limitations, providing users with a complete view of the board without any loss of information.

Overall, managing the printing of the board representation required addressing challenges related to board dimensions, formatting and alignment, handling different element types, and managing console window limitations. By implementing dynamic loops, proper formatting techniques, conditional statements, and sectioned printing, I successfully resolved these challenges and achieved an accurate and visually appealing representation of the board.

B) Defining Board and Square Classes:

I defined the relationship between the Board and Square objects by establishing a comprehensive and interconnected structure that reflects the characteristics and behaviors of a chessboard.

Firstly, I created a Board class that serves as the main container for the chessboard. This class has an internal representation of the 64 squares that make up the board, typically implemented as an array or a grid. Each square is then represented by an individual Square object.

To ensure the proper association between the Board and Square objects, I employed a composition relationship. This means that the Board class contains an array or collection of Square objects, which represents the collection of squares on the board. The composition relationship allows for a "whole-part" relationship between the Board and Square, where the Board is composed of multiple Square instances.

With this composition relationship in place, the Board class can easily access and manage the individual Square objects. This enables operations such as retrieving the state of a specific square, updating the position of game pieces, validating moves, and handling game logic. The Board class acts as a controller or manager for the Square objects, coordinating their interactions and ensuring the integrity of the chessboard structure.

On the other hand, the Square objects encapsulate the properties and behaviors associated with each square. They may contain attributes like position coordinates, color, and occupancy status. Additionally, Square objects may have methods to handle operations specific to a square, such as determining if it is under attack, calculating available moves, or updating the piece occupying the square.

By utilizing the composition relationship between the Board and Square objects, I establish a hierarchical structure that mirrors the real-world concept of a chessboard. This design approach promotes modularity, flexibility, and maintainability. It allows for efficient management of the squares on the board and facilitates the implementation of various chess game functionalities.

C) Implementing methods of Board and Square Classes:

ChessBoard Class:

1. isGameEnded()
- Determines if the game has ended by counting the number of remaining pieces.
- No parameters.
- Returns a boolean value indicating whether the game has ended.
- Iterates over each square on the board, counts the number of remaining pieces for each color, and checks if either color has no pieces remaining.

2. isWhitePlaying()
- Checks if it is currently the white player's turn.
- No parameters.
- Returns a boolean value indicating if it is the white player's turn.
- Simply returns the value of the static variable “whitePlaying".

3. initialize()
- Initializes the chessboard and sets up the pieces.
- No parameters.
- No return value.
- Creates Square objects for each position on the board and assigns them to the 2D array "board". Then creates and places the pieces on the board based on the starting positions.

4. toString()
- Represents the chessboard as a string for display.
- No parameters.
- Returns a string representation of the chessboard.
- Builds a string representation of the chessboard by iterating over the squares and appending the appropriate characters for each piece or empty square.

5. getBoard()
- Gets the entire chessboard as a 2D array of Square objects.
- No parameters.
- Returns a 2D array of Square objects representing the chessboard.
- Returns the "board" instance variable.

6. getPieceAt(String from)
- Gets the piece at a specific position on the chessboard.
- Parameter: from - the position in chess notation (e.g., "e4").
- Returns the Piece object at the specified position.
- Converts the chess notation to row and column coordinates, validates the input, and retrieves the piece from the corresponding square on the board.

7. getSquareAt(String to)
- Gets the square at a specific position on the chessboard.
- Parameter: to - the position in chess notation (e.g., "e4").
- Returns the Square object at the specified position.
- Converts the chess notation to row and column coordinates, validates the input, and retrieves the square from the corresponding location on the board.

8. getSquaresBetween(Square location, Square targetLocation)
- Gets the squares between two given squares.
- Parameters: location - the starting square, targetLocation - the ending square.
- Returns an array of Square objects representing the squares between the starting and ending squares.
- Checks if the squares are on the same row, column, or diagonal. If they are, it calculates the distance between them and creates an array of squares based on the distance. The array is populated by iterating from the starting square to the ending square.

9. nextPlayer()
- Updates the turn to the next player.
- No parameters.
- No return value.
- Inverts the value of the static variable "whitePlaying" to switch turns between players.

Square Class:

1. isAtLastRow(int color)
- Checks if the square is at the last row for a given color.
- Parameter: color - the color of the pieces (ChessBoard.WHITE or ChessBoard.BLACK).
- Returns a boolean value indicating if the square is at the last row for the specified color.
- Compares the row of the square with 7 for white and 0 for black.
2. isEmpty()
- Checks if the square is empty (no piece).
- No parameters.
- Returns a boolean value indicating if the square is empty.
- Checks if the "piece" instance variable is null.
3. isAtSameColumn(Square s)
- Checks if the square is in the same column as another square.
- Parameter: s - the other square to compare with.
- Returns a boolean value indicating if the squares are in the same column.
- Compares the column of the current square with the column of the other square.

4. getRowDistance(Square s)
- Calculates the row distance between this square and another square.
- Parameter: s - the other square to calculate the row distance with.
- Returns the absolute difference in row values between the two squares.
- Subtracts the row of the other square from the row of the current square and takes the absolute value.

5. getColDistance(Square s)
- Calculates the column distance between this square and another square.
- Parameter: s - the other square to calculate the column distance with.
- Returns the absolute difference in column values between the two squares.
- Subtracts the column of the other square from the column of the current square and takes the absolute value.

6. isNeighborColumn(Square s)
- Checks if the square is a neighbor column to another square.
- Parameter: s - the other square to compare with.
- Returns a boolean value indicating if the squares are neighbor columns.
- Calculates the column distance between the two squares and checks if it is equal to 1.

7. getBoard()
- Gets the chessboard associated with this square.
- No parameters.
- Returns the ChessBoard object associated with this square.
- Returns the "board" instance variable.

8. getPiece()
- Gets the piece on this square.
- No parameters.
- Returns the Piece object on this square.
- Returns the "piece" instance variable.

9. setPiece(Piece piece)
- Sets the piece on this square.
- Parameter: piece - the Piece object to set on this square.
- No return value.
- Sets the "piece" instance variable to the specified piece.

10. putNewQueen(int color)
- Promotes the pawn in this square to a new queen of the specified color.
- Parameter: color - the color of the new queen (ChessBoard.WHITE or ChessBoard.BLACK).
- No return value.
- Sets the "piece" instance variable to a new Queen object of the specified color.

11. clear()
- Clears the square (removes the piece).
- No parameters.
- No return value.
- Sets the "piece" instance variable to null.

D) Defining Piece Hierarchy:

In the Piece hierarchy, the Main class benefits from polymorphism by allowing it to work with different types of pieces through a common interface. This means that the Main class can treat different pieces uniformly without needing to know the specific type of each piece. It simplifies the code and makes it more flexible, as new types of pieces can be added without modifying the Main class.

In the Piece hierarchy, the methods and classes that can be defined as abstract depend on the specific design and requirements. Generally, abstract classes are used when there are common behaviors and attributes shared by multiple subclasses, but the exact implementation may vary. Abstract methods are declared in abstract classes but have no implementation and must be overridden by the concrete subclasses.

In my implementation, I would consider making the Piece class itself abstract if it provides common attributes or behaviors for all types of pieces, but should not be instantiated directly. Concrete subclasses such as Pawn, Rook, or Bishop would extend the Piece class and provide their specific implementations.

Regarding code reuse, there can be code reuse in the implementation of the Piece hierarchy. Common behaviors and attributes can be defined in the abstract classes, reducing the need for redundant code in the concrete subclasses. By inheriting from the abstract classes, the concrete subclasses automatically inherit and reuse the shared code. Additionally, if there are methods or behaviors that are common among multiple concrete subclasses, they can be implemented in the abstract classes, promoting code reuse and avoiding duplication.

E) Implementing methods in Piece Hierarchy:

Bishop Class:

public boolean canMove(String destination)

- The method checks whether the Bishop can move to the specified destination square.
- Parameter: destination - the location where the Bishop is intended to move.
- Returns a boolean value indicating whether the Bishop can move to the destination square.
- The method obtains the target location based on the provided destination. It checks if there are any squares between the current location and the target location. If squares exist, it iterates through them and returns false if any of the squares are not empty, indicating an obstruction. If no obstructions are found, the method checks if the target location is either empty or contains an opponent's piece, allowing the Bishop to move. Finally, it returns false if the Bishop is not on a diagonal path.

public void move(String destination)

- The method moves the Bishop to the specified destination square.
- Parameter: destination - the location where the Bishop is intended to move.
- The method doesn't return any value.
- The target location is obtained based on the provided destination. The Bishop is set as the piece in the target location, replacing any existing piece. The current location is cleared, and the new location of the Bishop is set as the target location. Finally, the move is considered complete, and it switches to the next player's turn.

public String toString()

- The method returns a string representation of the Bishop.
- The method doesn't have any parameters.
- It returns a string representation of the Bishop, using 'B' for a white Bishop and 'b' for a black Bishop.
- The method checks the color of the Bishop and returns 'B' if the color is white (ChessBoard.WHITE) or 'b' if the color is black.

King Class:

1. canMove(String destination)
- It checks if the King can move to the specified destination.
- Parameter: destination - a string representing the target location.
- Return: A boolean value indicating whether the King can move to the destination.
- Implementation: It calculates the row and column distance between the current location and the target location. If the distance is within 1 row and 1 column away and the target location is either empty or occupied by an opponent's piece, it returns true; otherwise, it returns false.
2. move(String destination)
- It moves the King to the specified destination.
- Parameter: destination - a string representing the target location.
- Return: None (void).
- Implementation: It gets the target location based on the destination string. It sets the current piece to the target location, clears the current location, updates the current location to the target location, and moves to the next player's turn.
3. toString()
- It returns a string representation of the King piece.
- Parameter: None.
- Return: A string representation of the King piece ("K" for white King and "k" for black King).
- Implementation: It checks the color of the King and returns the corresponding string representation ("K" for white and "k" for black).

Knight Class:

- canMove(String destination):
  - Determines whether the knight can move to the specified destination.
  - destination: the target location represented as a string.
  - Returns true if the move is valid for a knight (2 squares in one direction and 1 square in the other), and the target location is empty or has a piece of a different color. Otherwise, returns false.
  - The method calculates the distance between the current location and the target location, checks if the move is valid for a knight, and verifies the color of the piece in the target location.
- move(String destination):
  - Moves the knight to the specified destination.
  - destination: the target location represented as a string.
  - The method sets the knight as the piece in the target location, clears the current location of the knight, updates the knight's location to the target location, and moves to the next player's turn.
- toString():
  - Returns a string representation of the knight.
  - Represents the knight as "N" for white or "n" for black.

Pawn class:

1. public boolean canMove(String to)
    - Method functionality: Checks if the pawn can move to the specified square.
    - Parameters: to - the target square represented as a string.
    - Returns: true if the pawn can move to the target square, false otherwise.
    - Implementation: The method checks various conditions based on the pawn's color, current position, and the target square. It verifies if the move is valid for a pawn, considering forward movement, capturing diagonally, and special rules for the initial move.
2. public void move(String to)
    - Method functionality: Moves the pawn to the specified square.
    - Parameters: to - the target square represented as a string.
    - Returns: None.
    - Implementation: The method sets the pawn's target location, handles promotion to a queen if the pawn reaches the last row, updates the current location, clears the previous location, and advances the turn to the next player.
3. public String toString()
    - Method functionality: Returns a string representation of the pawn.
    - Parameters: None.
    - Returns: A string representation of the pawn: "P" for white pawns and "p" for black pawns.

Queen Class:

1. canMove(String destination)
    - What does the method do? Checks if the Queen can move to the specified destination square.
    - Parameter: destination - the target square where the Queen is trying to move.
    - Returns: boolean - true if the Queen can move to the destination, false otherwise.
    - How did you implement the functionality? The method obtains the targetLocation square based on the destination string. It then retrieves the squaresBetween the current location and the target location using the getSquaresBetween method from the board. If there are squares between the current location and the target location, the method checks if any of those squares are occupied (indicating an obstruction). If there are no obstructions, it checks if the target location is either empty or occupied by an opponent's piece to determine if the move is valid.
2. move(String destination)
    - What does the method do? Moves the Queen to the specified destination square.
    - Parameter: destination - the target square where the Queen should be moved.
    - Returns: None (void).
    - How did you implement the functionality? The method obtains the targetLocation square based on the destination string. It sets the current piece (the Queen) as the piece on the targetLocation square. It clears the current location of the Queen, sets the new location to the targetLocation, and advances the game to the next player.

3. toString()
    - What does the method do? Returns the string representation of the Queen.
    - Parameter: None.
    - Returns: String - the string representation of the Queen.
    - How did you implement the functionality? The method checks the color of the Queen (ChessBoard.WHITE for white, ChessBoard.BLACK for black) and returns "Q" for white Queen or "q" for black Queen as the string representation.

Rook class:

1. public boolean canMove(String destination):
    - Method functionality: Checks if the rook can move to the specified destination.
    - Parameter: destination - the target location where the rook is intended to move.
    - Return: true if the rook can move to the destination, false otherwise.
    - Implementation: Retrieves the target location on the board based on the provided destination. Obtains the squares between the current location of the rook and the target location. If there are squares between them, checks if any square is occupied. Returns false if any square is occupied, indicating that the rook cannot move. Otherwise, checks if the target location is empty or occupied by an opponent's piece and returns the result accordingly.
2. public void move(String destination):
    - Method functionality: Moves the rook to the specified destination.
    - Parameter: destination - the target location where the rook is intended to move.
    - Return: None.
    - Implementation: Retrieves the target location on the board based on the provided destination. Sets the rook as the piece on the target location, clears the current location of the rook, updates the location to the target location, and moves to the next player's turn.
3. public String toString():
    - Method functionality: Returns the string representation of the rook based on its color.
    - Parameter: None.
    - Return: A string representation of the rook. It returns "R" if the rook is white and "r" if it is black.
    - Implementation: Checks the color of the rook and returns the corresponding string representation.