棋CHESS GAME (JAVA PROJECT)

Made By :	**小摩 / IBRAHIM MOUNIR**	ID : **92337155E125**
and :	**哈德逊**	ID : **92337155E110**
and :	**阿布巴卡**	ID : **92115855E106**

To be Reviewed by the Professor : 凤琼
Subject : **JAVA Programming**

Project Overview:

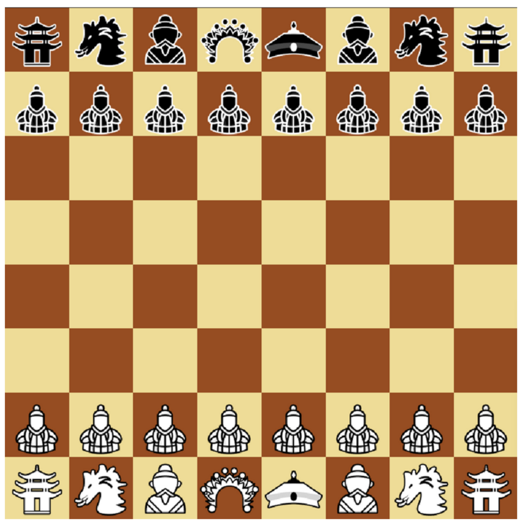
This project reimagines the traditional chess game with a unique twist—each piece is designed with elements of **Chinese culture and history**. The game features a rich visual style that blends traditional Chinese aesthetics with the classic gameplay of chess.

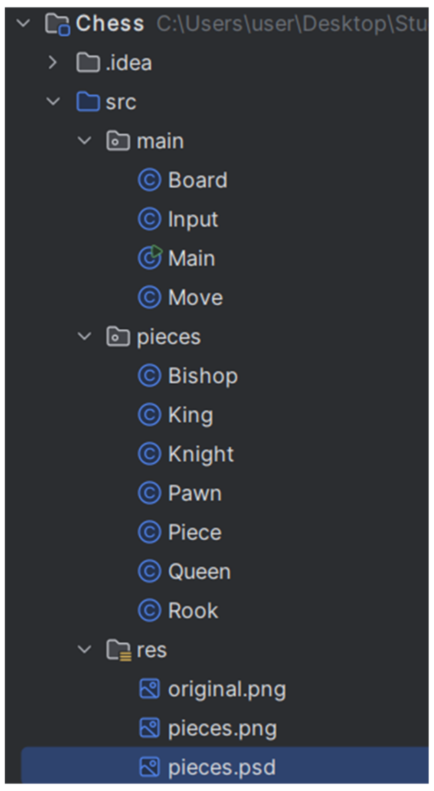
Features:

- **Graphical User Interface (GUI)**: Built with Java `Swing`, the game offers an interactive and visually rich chessboard, showcasing pieces inspired by Chinese culture and history.
- **Chinese-Themed Chess Pieces**: The standard chess pieces (King, Queen, Rook, Bishop, Knight, Pawn) are reinterpreted with Chinese cultural symbols, such as ancient warriors, generals, and mythical creatures.
- **Game Logic**: The project includes logic for piece movement, turn-based gameplay, checks, and checkmate, following standard chess rules.
- **Board Layout and Interaction**: The game board is structured with a traditional 8×8 grid, and players can interact with the pieces using mouse-driven controls, similar to a real-life chessboard.

PROJECT CLASSES :

Classes & Algorithm logic





```
Chess  C:\Users\user\Desktop\Stu
  .idea
  src
    main
      Board
      Input
      Main
      Move
    pieces
      Bishop
      King
      Knight
      Pawn
      Piece
      Queen
      Rook
    res
      original.png
      pieces.png
      pieces.psd
```

Main Class:

`Main.java`

This is the entry point of the Chess game application. The `Main` class is responsible for initializing the graphical interface and displaying the game board.

Key Components:

- **JFrame**: The main window of the application is created using `JFrame`. This window is customized with a black background and a grid layout.
- **Board**: The game board, where the chess pieces are displayed, is encapsulated in the `Board` class. It is added to the frame to render the game layout.
- **GridBagLayout**: The layout manager used to position the `Board` component in the frame.
- **Minimum Size**: The window size is set to a minimum of 1200×800 pixels to ensure the board is clearly visible.
- **Centering the Window**: The frame is positioned in the center of the screen using `frame.setLocationRelativeTo(null)`.

Code:

```
package main;

import javax.swing.*;
import java.awt.*;

public class Main {
    public static void main(String[] args) {
        // Create the main window (JFrame)
        JFrame frame = new JFrame();
        frame.getContentPane().setBackground(Color.black); // Set background color to black
        frame.setLayout(new GridBagLayout()); // Use GridBagLayout for flexible layout
        frame.setMinimumSize(new Dimension(1200, 800)); // Set the minimum size of the window
        frame.setLocationRelativeTo(null); // Center the window on the screen

        // Create the game board and add it to the window
        Board board = new Board();
        frame.add(board);

        // Make the window visible
        frame.setVisible(true);
    }
}
```

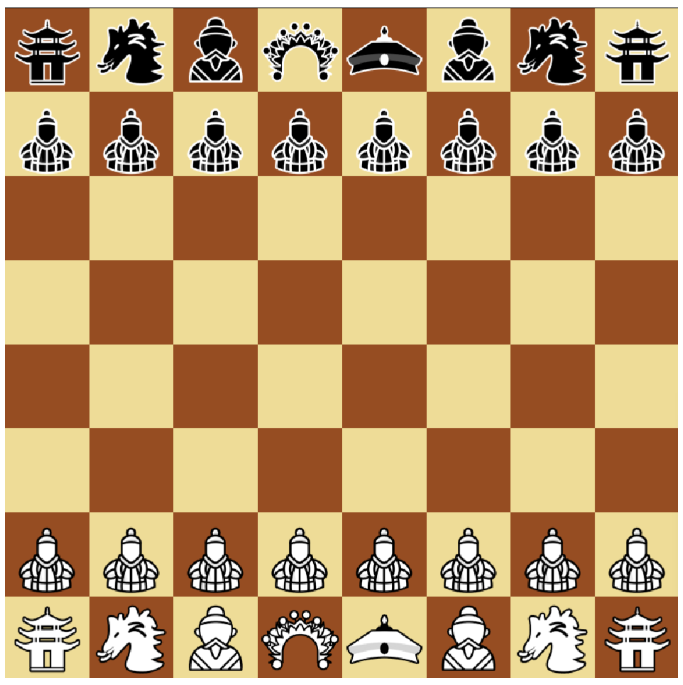
Breakdown:

1. **JFrame Creation**: Initializes the window using `JFrame` and sets properties like background color and minimum size.
2. **Board Initialization**: The `Board` object is created and added to the window. The board class handles the layout and rendering of the chess pieces.
3. **Visibility**: `frame.setVisible(true)` is called to ensure the window is displayed when the application runs.

Board Class :

`Board.java`

The `Board` class is responsible for representing the chessboard, managing the state of the game, and handling the interactions between the user and the game. It uses a `JPanel` to render the board and allows players to interact with it via mouse events.



Key Components:

- `isWhiteTurn` **(boolean)**: Indicates whether it's the white player's turn. If `true`, white moves; otherwise, black moves.
- `titleSize` **(int)**: The size (in pixels) of each square on the chessboard. The default is `95` pixels.
- `cols` **(int)** and `rows` **(int)**: Define the size of the board grid. Default values are both set to `8` for an 8×8 chessboard.
- `pieceList` **(ArrayList<Piece>)**: A list of all pieces currently on the board.
- `capturedWhitePieces` **(ArrayList<Piece>)** and `capturedBlackPieces` **(ArrayList<Piece>)**: Lists that hold the captured pieces for both the white and black sides.
- `selectedPiece` **(Piece)**: The currently selected piece (if any), used to highlight and move the piece on the board.
- `input` **(Input)**: A `MouseListener` and `MouseMotionListener` instance that captures mouse events for interacting with the board.
- `isCheck` **(boolean)**: A flag indicating if the current player's king is in check.
- `isCheckmate` **(boolean)**: A flag indicating if the game is in a checkmate state.
- `statusLabel` **(JLabel)**: A UI component used to display game status messages (e.g., check, checkmate).
- `quitButton` **(JButton)**: A button to quit the game.
- `replayButton` **(JButton)**: A button to reset and replay the game.

Key Methods:

- **Constructor (**`Board()`**)**:
  - Initializes the board layout and adds the initial pieces to the board.

    - Sets up UI elements like buttons for quitting or replaying the game.
    - Registers mouse listeners to handle piece selection and movement.
- `loadPosition()`:
    - Sets up the initial state of the board by adding all chess pieces (Rooks, Knights, Bishops, Queens, Kings, and Pawns) in their correct starting positions.
- `makeMove(Move move)`:
    - Handles the movement of a selected piece on the board.
    - Updates the piece’s position and checks for captured pieces.
    - Verifies if the move results in a "check" or "checkmate".
- `capture(Move move)`:
    - Handles the capturing of an opponent's piece when a move results in a capture.
    - If a King is captured, the game ends with the opposing player winning.
- `isValidMove(Move move)`:
    - Checks if a move is valid according to the rules of chess (e.g., piece movement, turn validation, no collisions).
- `paintComponent(Graphics g)`:
    - Responsible for rendering the chessboard and pieces.
    - Highlights valid moves for the selected piece in green.
    - Paints the board with alternating dark and light squares, and positions the pieces correctly.
- `checkForCheck()`:
    - Checks if the current player's King is in check (i.e., threatened by an opponent's piece).
    - This method is invoked after each move to determine whether the current player is in check.
- `isCheckmate()`:
    - Checks if the game has reached a checkmate state, where the current player's King is in check and cannot escape.
    - If the King cannot move to a safe square or any other piece can block the check, the game ends.

Piece Class :

`Piece.java` :

The `Piece` class serves as the base class for all chess pieces, encapsulating the shared properties and behaviors of all pieces. Each piece has a position on the board (`col`, `row`), and a graphical representation (`sprite`). The class provides default implementations for movement validation and collision checking that can be overridden by specific pieces, like the `Bishop`, `King`, `Pawn`, etc.

Key Fields:

- `col`, `row`: The piece’s position on the chessboard (columns and rows are indexed from 0 to 7).
- `xPos`, `yPos`: The pixel coordinates for the piece's position on the screen.
- `isWhite`: A boolean flag indicating whether the piece is white or black.
- `name`: The name of the piece (e.g., "Bishop", "King").
- `sheet`: A `BufferedImage` representing the image sheet containing all the chess piece sprites.
- `sprite`: A specific image for the piece, extracted from the `sheet` and resized to fit the board's tile size.

- `board`: A reference to the `Board` class, which holds the state of the game.

Key Methods:

- **Constructor (** `Piece(Board board)` **)**: Initializes the piece with its board reference and assigns its position and sprite image.
- `isValidMovement(int col, int row)`: Checks if a proposed move to the specified coordinates is valid. The base class implementation always returns `true`, but this method is overridden in subclasses to enforce piece-specific movement rules.
- `moveCollidesWithPiece(int col, int row)`: Determines if the move collides with another piece on the board. The base class implementation always returns `false`, but this is also overridden in specific pieces like the `Bishop`.
- `paint(Graphics2D g2d)`: Renders the piece on the board at its `xPos` and `yPos`.

Bishop.java (Subclass of Piece):

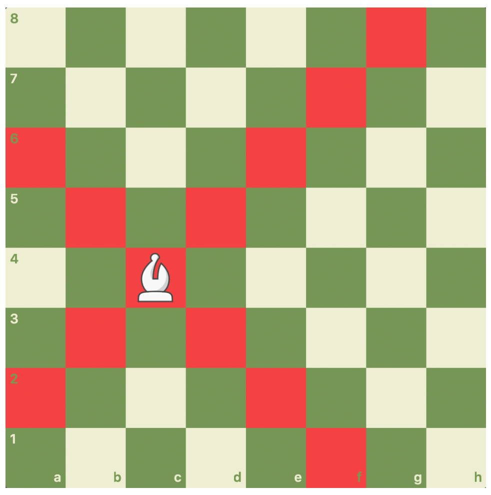
The `Bishop` class extends the `Piece` class and implements the specific rules for how a bishop moves on the chessboard. Bishops can move diagonally any number of squares, as long as they are not blocked by other pieces.

Key Differences in `Bishop` Class:

- **Constructor (** `Bishop(Board board, int col, int row, boolean isWhite)` **)**:
  - Initializes the bishop with its position, color, and assigns the correct sprite (white or black).
  - The `sprite` is scaled to the size of the board tiles using `getSubimage()` and `getScaledInstance()`.
- `isValidMovement(int col, int row)`:
  - The bishop can only move diagonally, so this method returns `true` if the difference between the current and destination row and column is equal (`Math.abs(this.col - col) == Math.abs(this.row - row)`).
- `moveCollidesWithPiece(int col, int row)`:
  - Checks if the bishop's movement collides with another piece.
  - The method checks each diagonal direction (up-left, up-right, down-left, down-right) for pieces blocking the way by iterating over all squares between the starting and destination square.
  - If a piece is found along the path, it returns `true` to indicate a collision.

Code: `Bishop.java`

```
java
Copy code
package pieces;

import main.Board;
import java.awt.image.BufferedImage;

public class Bishop extends Piece {
    public Bishop(Board board, int col, int row, boolean isWhite) {
        super(board);
        this.col = col;
        this.row = row;
        this.xPos = col * board.titleSize;
        this.yPos = row * board.titleSize;

        this.isWhite = isWhite;
        this.name = "Bishop";

        // Load the correct sprite for the bishop
        this.sprite = sheet.getSubimage(2 * sheetScale, isWhite ? 0 : sheetScale, sheetScale,
sheetScale)
                            .getScaledInstance(board.titleSize, board.titleSize, BufferedImag
e.SCALE_SMOOTH);
    }

    @Override
    public boolean isValidMovement(int col, int row) {
        // Bishops move diagonally, so the absolute difference between the row and column mus
t be equal
        return Math.abs(this.col - col) == Math.abs(this.row - row);
    }

    @Override
    public boolean moveCollidesWithPiece(int col, int row) {
        // Check if the move collides with a piece along the diagonal
        // Up-left
        if (this.col > col && this.row > row)
            for (int i = 1; i < Math.abs(this.col - col); i++)
                if (board.getPiece(this.col - i, this.row - i) != null)
                    return true;
        // Up-right
        if (this.col < col && this.row > row)
            for (int i = 1; i < Math.abs(this.col - col); i++)
                if (board.getPiece(this.col + i, this.row - i) != null)
                    return true;
        // Down-left
        if (this.col > col && this.row < row)
            for (int i = 1; i < Math.abs(this.col - col); i++)
                if (board.getPiece(this.col - i, this.row + i) != null)
                    return true;
        // Down-right
        if (this.col < col && this.row < row)
            for (int i = 1; i < Math.abs(this.col - col); i++)
                if (board.getPiece(this.col + i, this.row + i) != null)
                    return true;

        return false; // No collision
    }
}
```



Piece Movement Logic:

- Each piece has its own unique movement rules, but all pieces inherit from the `Piece` base class. Subclasses override the `isValidMovement()` and `moveCollidesWithPiece()` methods to define the specific movement behavior for each type of piece (e.g., Knight, Queen, King, etc.).

Summary of OTHER Piece Subclasses:

1. King Class (`King.java`):

The King is a special piece that can move one square in any direction (vertically, horizontally, or diagonally).

Key Methods:

- `isValidMovement(int col, int row)` :
  - The King can move only one square in any direction. This method checks if the destination square is within one square horizontally, vertically, or diagonally from the current position.
- `moveCollidesWithPiece(int col, int row)` :
  - The King cannot move to a square that is already occupied by a friendly piece (same color). This method checks if the target square contains a piece of the same color and returns `true` if there's a collision.

2. Knight Class (`Knight.java`)

The Knight moves in an "L" shape: two squares in one direction and then one square perpendicular to that. It is the only piece that can "jump" over other pieces.

Key Methods:

- `isValidMovement(int col, int row)` :
  - The Knight's move is validated by checking if the absolute product of the difference in columns and rows equals 2. This ensures the move is in the L-shape pattern.
- `moveCollidesWithPiece(int col, int row)` :
  - The Knight can move to a square occupied by an opponent's piece or an empty square. This method checks if the target square contains a friendly piece and returns `true` if there's a collision.

3. Pawn Class (`Pawn.java`)

The Pawn is the only piece that moves differently depending on whether it is its first move. It moves one square forward, but on its first move, it can move two squares forward. Pawns also capture pieces diagonally and have a special rule called "En Passant."

Key Methods:

- `isValidMovement(int col, int row)` :

    - The Pawn's movement depends on whether it's moving one square or two squares forward. It also allows for diagonal captures and the special En Passant move, where a pawn can capture an opponent's pawn that just moved two squares forward.
- `moveCollidesWithPiece(int col, int row)`:
    - The Pawn checks for collisions when moving one or two squares forward. It also ensures there's no piece blocking its path when moving two squares forward. For captures, the Pawn checks if an enemy piece is present on the target square.

4. Queen Class (`Queen.java`)

The Queen is the most powerful piece on the board, capable of moving any number of squares horizontally, vertically, or diagonally.

Key Methods:

- `isValidMovement(int col, int row)`:
    - The Queen can move any number of squares in any direction (straight or diagonal). This method checks if the movement is along the same row, column, or diagonal.
- `moveCollidesWithPiece(int col, int row)`:
    - The Queen's movement is checked in four directions (left, right, up, down, and diagonals). It checks if there is any piece blocking its path before making the move.

5. Rook Class (`Rook.java`)

The Rook moves any number of squares along a row or column, but not diagonally.

Key Methods:

- `isValidMovement(int col, int row)`:
    - The Rook can move horizontally or vertically. This method checks if the column or row of the start and destination positions are the same.
- `moveCollidesWithPiece(int col, int row)`:
    - The Rook checks for collisions along its path by iterating over the squares in the direction it is moving (left, right, up, or down) to ensure there are no pieces in the way.

Key Observations:

- **Movement Validation**: Each subclass overrides the `isValidMovement` method to enforce the specific movement rules for that piece type.
- **Collision Detection**: The `moveCollidesWithPiece` method ensures that no piece can land on a square that is occupied by a friendly piece. Additionally, it checks whether a path is blocked by another piece (for sliding pieces like the Queen, Rook, and Bishop).
- **Piece Initialization**: Each subclass has a constructor that initializes the piece with its position, color, and sprite. The sprite for each piece is retrieved from a shared image sheet (`pieces.png`) and scaled to fit the board tiles.

Input Class

Purpose:

The `Input` class is responsible for handling mouse interactions (clicking, dragging, and releasing) to select and move chess pieces on the board. It listens for mouse events and updates the position of the selected piece accordingly.

Key Methods:

1. `mousePressed(MouseEvent e)`:
   - This method is triggered when the mouse is pressed.
   - It calculates the column (`col`) and row (`row`) based on the mouse's x and y coordinates (divided by the tile size).
   - If there is a piece at the calculated position, it sets `board.selectedPiece` to that piece, indicating that the user has selected it.

```
@Override
public void mousePressed(MouseEvent e) {
    int col = e.getX() / board.titleSize;
    int row = e.getY() / board.titleSize;

    Piece pieceXY = board.getPiece(col, row);
    if(pieceXY != null){
        board.selectedPiece = pieceXY;
    }
}
```

2. `mouseDragged(MouseEvent e)`:
   - This method is called while the mouse is being dragged. It moves the `selectedPiece` based on the new mouse position.
   - The piece's new position is adjusted by subtracting half the tile size (`board.titleSize / 2`) to center the piece on the cursor.
   - It then triggers a repaint of the board to visually update the piece's position.

```
@Override
public void mouseDragged(MouseEvent e) {
    if(board.selectedPiece != null){
        board.selectedPiece.xPos = e.getX() - board.titleSize / 2;
        board.selectedPiece.yPos = e.getY() - board.titleSize / 2;

        board.repaint();
    }
}
```

3. `mouseReleased(MouseEvent e)`:
   - This method is triggered when the mouse button is released.
   - It calculates the target square (`col`, `row`) based on the mouse's position.
   - It creates a `Move` object representing the potential move from the piece's original position to the new position.
   - If the move is valid (as determined by the `board.isValidMove()` method), it applies the move by calling `board.makeMove(move)`.
   - If the move is invalid, it reverts the piece to its original position.
   - Finally, it sets `selectedPiece` to `null` and repaints the board to reflect the changes.

```
@Override
public void mouseReleased(MouseEvent e) {
    int col = e.getX()/board.titleSize;
    int row = e.getY()/board.titleSize;

    if(board.selectedPiece != null){
        Move move = new Move(board, board.selectedPiece, col, row);

        if(board.isValidMove(move)){
            board.makeMove(move);
        } else {
            board.selectedPiece.xPos = board.selectedPiece.col * board.titleSize;
            board.selectedPiece.yPos = board.selectedPiece.row * board.titleSize;
        }
    }
    board.selectedPiece = null;
    board.repaint();
}
```

Move Class

Purpose:

The `Move` class represents a move that a piece makes on the board. It holds information about the starting position, the target position, the piece being moved, and any captured piece.

Key Fields:

1. **`oldCol`**, **`oldRow`**, **`newCol`**, **`newRow`**:
    - These fields store the coordinates of the piece's initial position (`oldCol`, `oldRow`) and the target position (`newCol`, `newRow`).
2. **`piece`**:
    - This field stores the piece that is being moved.
3. **`capture`**:
    - This field stores the piece that is captured (if any) during the move. It's fetched from the `Board` using the target position (`newCol`, `newRow`).

Constructor:

The constructor initializes the move object by capturing the starting and ending positions and the relevant piece data.

```
public Move(Board board, Piece piece, int newCol, int newRow){
    this.oldCol = piece.col;
    this.oldRow = piece.row;
    this.newCol = newCol;
    this.newRow = newRow;

    this.piece = piece;
    this.capture = board.getPiece(newCol, newRow);
}
```

This constructor provides the necessary data for a move:

- **`oldCol`** and **`oldRow`** store the initial coordinates of the piece.
- **`newCol`** and **`newRow`** store the target coordinates.
- The **`piece`** field stores the piece being moved, and **`capture`** stores any captured piece at the target position.

How They Work Together:

1. **Interaction with the Board**:
    - When the user clicks on a square, `mousePressed` checks if a piece exists at that position. If so, it selects the piece by assigning it to `selectedPiece` on the `Board`.
    - As the user drags the piece, `mouseDragged` updates the position of `selectedPiece` to follow the mouse, allowing the user to visually move the piece.

- Upon releasing the mouse (`mouseReleased`), the target position is calculated. A `Move` object is created with the starting and target coordinates and the piece being moved.
- The `isValidMove` method on the `Board` class is called to check if the move is valid according to the rules of chess.
  - If valid, `makeMove(move)` applies the move, updating the piece's position and handling any captured pieces.
  - If the move is invalid, the piece is returned to its original position.

2. **Validation and Execution**:
   - `isValidMove` should perform a range of checks, such as ensuring the piece's move is valid (based on its type) and checking for any collisions with other pieces (e.g., no blocking pieces).
   - If the move is valid, `makeMove` should update the board state by moving the piece and handling any captures.

Fixes and Features to Add:

1. **Pawn Promotion:**
   - Implement pawn promotion when a pawn reaches the last row. Allow the player to choose between a queen, rook, bishop, or knight.
   - Consider adding a UI prompt for this selection.
2. **En Passant:**
   - Implement the en passant rule, where a pawn can capture an opponent's pawn if it moves two squares forward from its starting position and lands next to the capturing pawn.
   - Ensure that the conditions for en passant are correctly checked (the target square and timing).
3. **Check and Checkmate Fixes:**
   - Fix the logic for detecting check and checkmate to ensure that the game correctly identifies when the king is in check or checkmate.
   - Review the movement rules and blocking pieces to ensure accuracy.
4. **Movement Bugs:**
   - Investigate and fix any issues related to piece movement, including miscalculations or incorrect rules (e.g., knights, castling, etc.).

UI Improvements:

1. **User Interface (UI):**
   - Create a visually appealing and intuitive UI with clear piece selection, move highlighting, and status indicators (e.g., turn indicator, check, checkmate).
   - Consider using JavaFX or Swing for enhanced UI elements.
2. **Undo and Redo:**
   - Implement an undo/redo functionality that allows players to revert or reapply their previous moves.
   - Store game states after each move to enable this feature.
3. **Captured Pieces:**
   - Display captured pieces on the sides of the board in a separate panel. You could categorize them by color and type (e.g., "White Captured" and "Black Captured").
   - Update the captured pieces list after every move and capture.