# Chess Game Rules

Chess is a two-player strategy board game that involves moving pieces on an black and white checkered 8x8 grid, also known as a chessboard. The game is played on a board with 64 squares, and each player starts with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The objective of the game is to checkmate the opponent's king, by putting the opponent’s king in a position where it is under threat and there is no way to escape capture.

The rules of a chess game include:

1. Starting Position:

* Each player sets up their pieces on the two rows closest to them.
* The back row (or rank) from left to right is usually set up as follows: rook, knight, bishop, queen, king, bishop, knight, rook.
* The front row is filled with pawns.

1. Piece Movement:

* **King**: Moves one square in any direction.
* **Queen**: Moves diagonally, horizontally, or vertically any number of squares.
* **Rook**: Moves horizontally or vertically any number of squares.
* **Bishop**: Moves diagonally any number of squares.
* **Knight**: Moves in an "L" shape: two squares in one direction and one square perpendicular.
* **Pawn**: Moves forward one square but captures diagonally. On its first move, a pawn has the option to move forward two squares.

1. Special Moves:

* **Castling**: A king can castle with a rook if neither piece has moved, there are no pieces between them, and the king is not in check. Castling has been implemented in the game proposed.
* **En Passant**: A pawn capturing move that can occur when a pawn moves two squares forward from its starting position. En Passant has been implemented in the game proposed

1. Check and Checkmate:

* When a king is under direct threat of capture, it is said to be in check.
* The checkmate function is not implemented into the game. Instead, the king has to be captured in order to win.

1. Draws:

* Draws can occur by a player forfeiting or reaching the limit of turns set by the user when setting up the game.

1. Timer

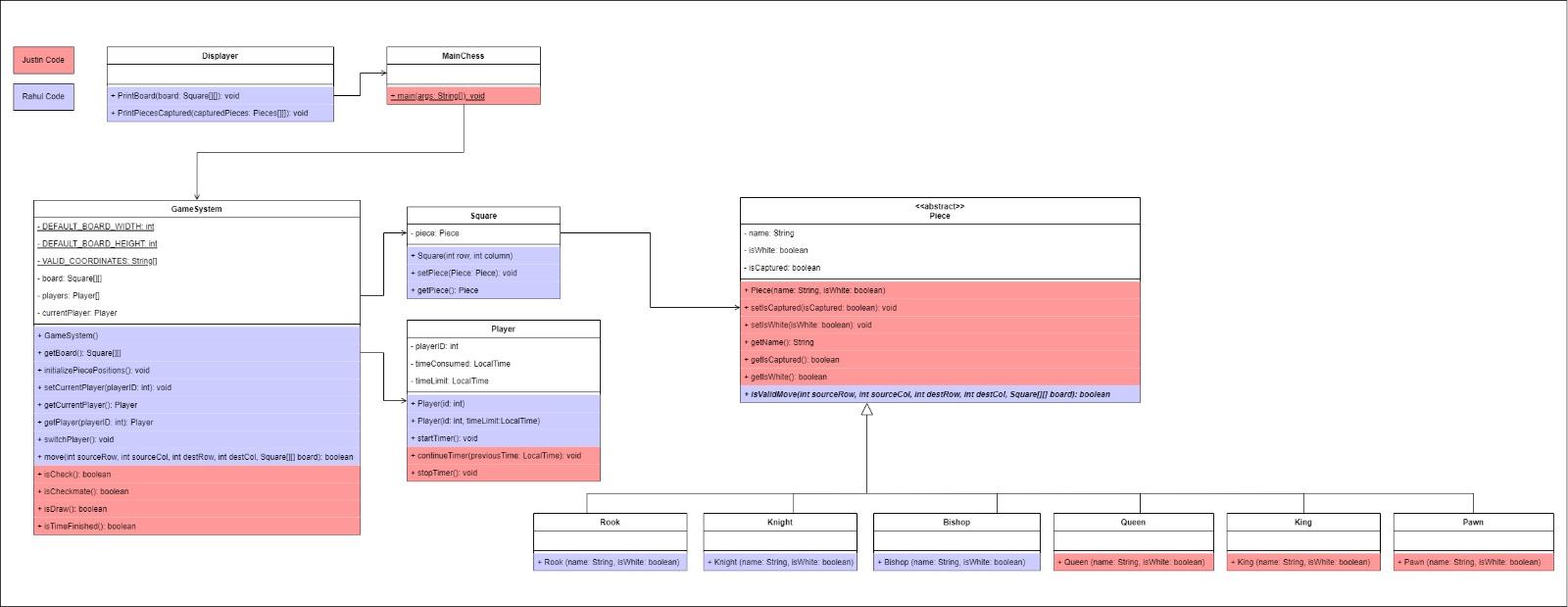
* The players are prompted to enter a time limit when the game is being set up. This time limit is the maximum amount of time a player has, similar to a chess clock. However, unlike a chess clock, whoever’s time runs out first loses.

# UML Diagram Drafts

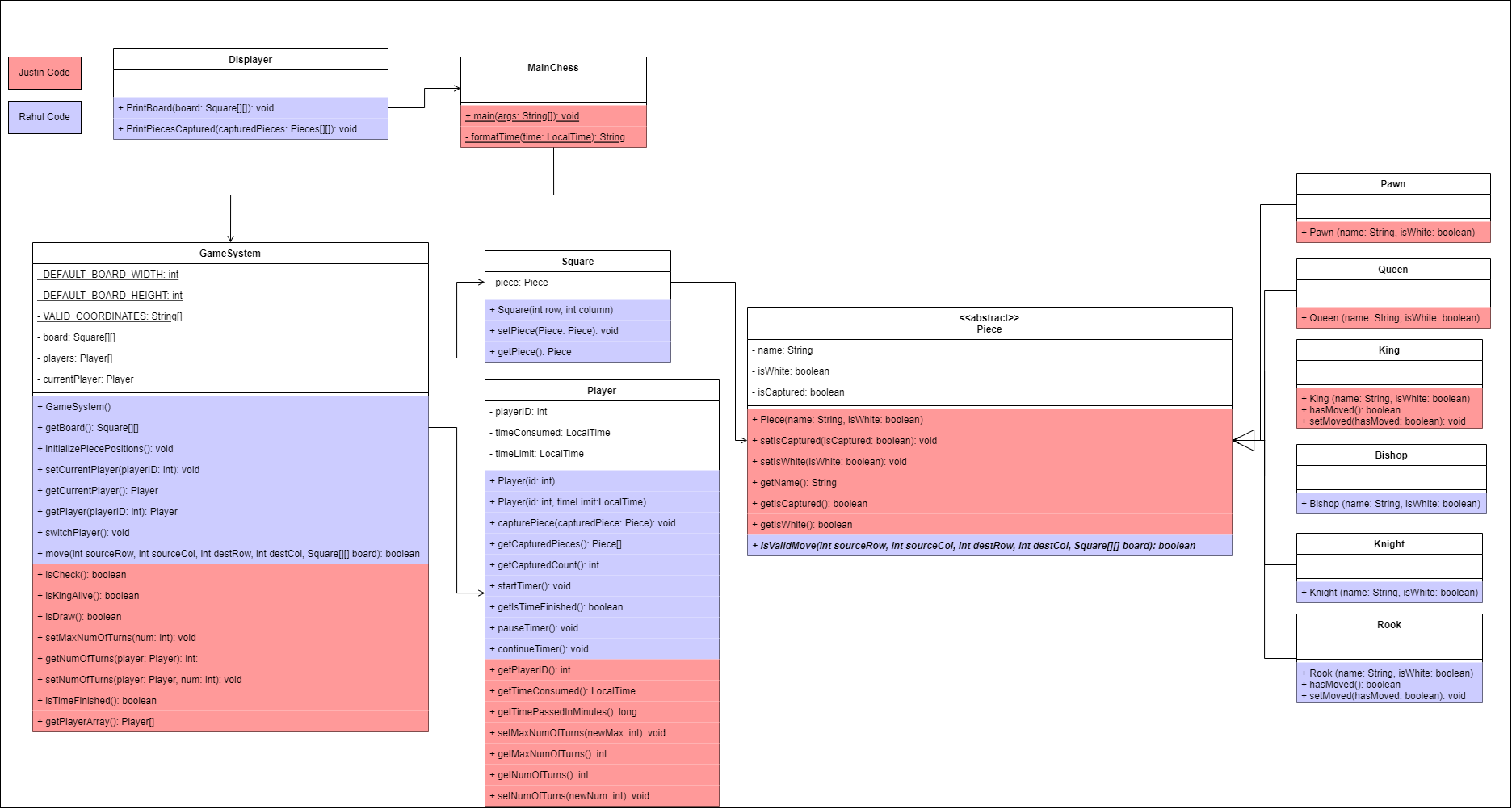
## 1st Draft

## 

## 2nd Draft



## Final Draft



# Class Implementation

1. Piece Class

* Purpose: This is an abstract class representing a generic chess piece. It contains common properties such as name, colour, and whether the piece is captured.
* Fields:
  + **String name:** Represents the name of the chess piece.
  + **boolean isWhite:** Indicates whether the piece is white.
  + **boolean isCaptured:** Indicates whether the piece is captured.
* Methods:
  + **Abstract boolean isValidMove(int sourceRow, int sourceCol, int destRow, int destCol, Square[][] board):** This method is declared abstract, meaning it must be implemented by subclasses. It checks if a move from the source square to the destination square is valid for the specific type of chess piece.

1. Rook Class

* Purpose: Represents a rook chess piece. Extends the Piece class and includes functionality specific to rooks, such as tracking whether the rook has moved.
* Fields:
  + **boolean hasMoved:** Indicates whether the rook has moved.
* Methods:
  + **boolean isValidMove(int sourceRow, int sourceCol, int destRow, int destCol, Square[][] board):** Checks if the move is valid for a rook. It verifies if the move is either horizontal or vertical and if the path is clear.
  + **boolean hasMoved():** Returns whether the rook has moved.
  + **void setMoved(boolean hasMoved):** Sets the moved status of the rook.

1. Queen Class

* Purpose: Represents a queen chess piece, extending the Piece class.Implements methods to check the validity of queen moves on a chessboard.
* Fields:
  + None specific to the queen class.
* Methods:
  + **boolean isValidMove(int sourceRow, int sourceCol, int destRow, int destCol, Square[][] board):** Checks if the move is valid for a queen. It verifies if the move is either horizontal, vertical, or diagonal and if the path is clear.

1. King Class

* Purpose: Represents a king chess piece, extending the Piece class. Includes functionality specific to kings, such as tracking whether the king has moved and checking the validity of its moves.
* Fields:
  + **boolean hasMoved:** Indicates whether the king has moved.
* Methods:
  + **boolean isValidMove(int sourceRow, int sourceCol, int destRow, int destCol, Square[][] board):** Checks if the move is valid for a king. It verifies if the move is a standard one-square move or a castling maneuver.
  + **boolean hasMoved():** Returns whether the king has moved.

1. Pawn Class

* Purpose: Represents a pawn chess piece, extending the Piece class. Implements methods to check the validity of pawn moves on a chessboard.
* Fields:
  + **boolean hasMoved:** Indicates whether the pawn has moved.
* Methods:
  + **boolean isValidMove(int sourceRow, int sourceCol, int destRow, int destCol, Square[][] board):** Checks if the move is valid for a pawn. It considers the current state of the chessboard and the specific rules for pawn movement.
  + **boolean hasMoved():** Returns whether the pawn has moved.

1. Knight Class

* Purpose: Represents a knight chess piece, extending the Piece class. Implements specific rules for the knight's movement on a chessboard.
* Fields:
  + None specific to the knight class.
* Methods:
  + **boolean isValidMove(int sourceRow, int sourceCol, int destRow, int destCol, Square[][] board):** Checks if the proposed move for the knight is valid based on the rules of chess. The knight moves in an L-shaped pattern, two squares in one direction and one square perpendicular to that direction, or vice versa.

1. Bishop Class

* Purpose: Represents a bishop chess piece, extending the Piece class. Includes methods to validate bishop moves on a chessboard.
* Fields:
  + None specific to the bishop class.
* **Methods: boolean isValidMove(int sourceRow, int sourceCol, int destRow, int destCol, Square[][] board):** Validates if the bishop can move from the source position to the destination position on the chessboard. The method checks if the move is diagonal and if there are no pieces in the diagonal path.

1. Square Class

* Purpose: Represents a square on a game board. Each square may contain a Piece.
* Fields:
  + **private Piece piece:** Represents the piece currently on the square.
* Methods:
  + **void setPiece(Piece piece):** Sets the piece on this square.
  + **Piece getPiece():** Gets the piece currently on this square. Returns the piece on the square, or null if no piece is present.

1. Player Class

* Purpose: Represents a player with a unique ID, a timer, and methods for managing time-related activities as well as checking the number of turns. Provides functionality for starting, stopping, pausing, and continuing a timer. Methods to retrieve the time consumed and the time passed in minutes.
* Fields:
  + **private int playerID:** The unique identifier for the player.
  + **private LocalTime timeConsumed:** The time consumed by the player, initialized to 0 hours and 0 minutes.
  + **private LocalTime timeLimit:** The time limit for the player's activities.
  + **private Timer timer:** The timer used to track player time.
  + **private boolean isTimerPaused:** Flag indicating whether the timer is paused.
  + **private int capturedCount:** The count of captured pieces.
  + **private Piece[] capturedPieces:** The array of captured pieces by the player.
  + **private int numOfTurns = 0:** The default number of turns completed for a user
  + **private int maxNumOfTurns = 50:** The default number of turns before a draw is reached.
* Methods:
* **public Player(int id):** Constructs a new player with the specified ID.
* **public Player(int id, LocalTime timeLimit):** Constructs a new player with the specified ID and time limit.
* **public void capturePiece(Piece capturedPiece):** Adds a captured piece to the player's array.
* **public Piece[] getCapturedPieces():** Gets the array of captured pieces.
* **public int getCapturedCount():** Gets the count of captured pieces.
* **public void startTimer():** Starts the timer for the player. Increments time consumed by one second at regular intervals.
* **public void stopTimer():** Stops the player's timer.
* **public void pauseTimer():** Pauses the player's timer.
* **public void continueTimer():** Continues the player's timer from where it was paused.
* **public LocalTime getTimeConsumed():** Gets the total time consumed by the player.
* **public long getTimePassedInMinutes():** Gets the total time passed in minutes.
* **public void setMaxNumOfTurns(int newMax):** Set the maximum number of turns
* **public int getMaxNumOfTurns():** Get the maximum number of turns
* **public int getNumOfTurns():** Get the current number of turns for a specified player
* **public void setNumOfTurns(int newNum):** Set a number of turns

1. Displayer Class

* Purpose: Provides methods to display the chessboard and captured pieces for players.
* Methods:
  + **void printBoard(Square[][] board):** Prints the current state of the chessboard, including piece names and positions. It formats the output to display each piece's name along with its position.
  + **void printCapturedPieces(Player[] players):** Prints the captured pieces for each player. It iterates over the players and displays the captured pieces for each, separating them by player ID.

1. GameSystem Class

* Purpose: Represents a chess game system with a game board, players, and methods for performing specific chess moves.
* Fields:
  + **public static int DEFAULT\_BOARD\_WIDTH:** The default width of the game board.
  + **public static int DEFAULT\_BOARD\_HEIGHT:** The default height of the game board.
  + **private Square[][] board:** Represents the game board as a two-dimensional array of squares.
  + **private Player[] players:** Represents the array of players participating in the game.
  + **private Player currentPlayer:** Represents the current player making a move.
* Methods:
  + **private void performKingSideCastling(int kingSourceCol, int kingDestCol, int rookSourceCol, int rookDestCol, int row):** Private method to perform King-side castling.
  + **private void performQueenSideCastling(int kingSourceCol, int kingDestCol, int rookSourceCol, int rookDestCol, int row):** Private method to perform Queen-side castling.
  + **private boolean isPathClear(int sourceRow, int sourceCol, int destRow, int destCol):** Private helper method to check if the path is clear between two squares horizontally or vertically.
  + **public GameSystem():** Default constructor for the GameSystem class. Initializes the game board, players, and sets the current player to the first player.
  + **public Square[][] getBoard():** Gets the current state of the game board.
  + **public void initializePiecePositions():** Initializes the positions of chess pieces on the game board at the beginning of the game.
  + **public void setCurrentPlayer(int playerID):** Sets the current player based on the provided player ID.
  + **public Player getCurrentPlayer():** Gets the current player making a move.
  + **public Player getPlayer(int playerID):** Gets the player object based on the provided player ID.
  + **public void switchPlayer():** Switches the current player between players[0] and players[1].
  + **public int move(int sourceRow, int sourceCol, int destRow, int destCol):** Moves a chess piece from the source square to the destination square on the game board. Utilizes movement rules of the pieces and handles various scenarios, returning specific codes to indicate the result of the move.
  + **public boolean isCheck()**: Gives true or false depending on whether the king of the current player is under threat
  + **public boolean isKingAlive()**: Gives true or false depending on whether the king of the current player is still on the board
  + **public boolean isDraw()**: Gives true or false depending on whether the number of turns has been reached for both players
  + **public void setMaxNumOfTurns(int num):** Set the upper limit of turns
  + **public void getNumOfTurns(Player player):** Get the current number of turns for a specified player.
  + **public void setNumOfTurns(Player player, int num):** Set the number of turns per specified playerID
  + **public boolean isTimeFinished():** Gives true or false depending on whether time given for a player has run out.
  + **public Player[] getPlayerArray():** Returns the Player[] array for use in MainChess

1. MainChess class

* Methods
  + **public static void main(String[] args):** The main program
  + **private static String formattime(LocalTime time):** Helper method to format LocalTime as HH:mm:ss

# Inheritance

A screenshot of a computer program

Description automatically generated

*Figure 1 (Piece Class)*

Figure 1 shows the Piece class which represents the parent class in our code. Here, all the child classes: Knight, Rook, Bishop, Pawn, Queen, and King have 3 main common attributes: name (for display purpose), colour (can also be either white or book or isWhite) and captured status. Furthermore, these attributes also inherit the isValidMove function which determines the validity of a move alongside the getters and setter functions included.

*A computer code with red and blue text

Description automatically generated*

*Figure 2 (Snippet of Pawn Class)*

Figure 2 shows a snippet of the Pawn class which inherits the attributes from the Piece class. As seen in the super() function call, only name and isWhite attributes are assigned. isCaptured variable is only assigned a value during the capturing or movement of another piece onto the Pawn piece. But for the time being, pawn is set isCaptured = false since it is automatically done by JAVA during compile time*.*

# Encapsulation

A screenshot of a computer program

Description automatically generated

*Figure 3*

Figure 3 shows a representation of an example on how encapsulation in the code looks like. Here we have the class Player which contains the attributes playerID, timeConsumed, timeLimit, timer, isTimerPaused, capturedCount. These variables have private visibility access meaning that they can only be accessed within its own class. This is a good practice to maintain to prevent other classes from freely modifying these variables.

These variables can be set or returned using getters and setter functions which have been implemented as well.

A screenshot of a computer program

Description automatically generated

*Figure 4 (Showing a snippet of getter functions)*

A screenshot of a computer program

Description automatically generated

*Figure 5 (Showing a snippet of setter functions)*

# Polymorphism

As seen in Figure 1, we know that inheritance is implemented. The only form of polymorphism implemented in our code is method overriding. This implementation is a combination of abstraction and polymorphism. Since every child class of Piece has its own movement style, example a pawn one step at a time, bishop diagonal and so on, it is important that we create an abstract method to check the validity of the movement of every child Piece. This is where isValidMove function steps in and creates method overriding.

A screenshot of a computer code

Description automatically generated

*Figure 6 (isValidMove for Pawn)*

A screen shot of a computer code

Description automatically generated

*Figure 7 (isValidMove for Knight)*

A screenshot of a computer code

Description automatically generated

*Figure 8 (isValidMove for Bishop)*

As seen by Figures 6, 7 and 8, the implementation of isValidMove differs from one child class to another. The pawn class checks for steps one piece at a time, while Knight checks for steps in an L-shape and Bishop checks the movement in its own diagonal path.