```java
package engine;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.PlayerColor;


public class Player {
   private final PlayerColor color;

   public Player(PlayerColor color){
      this.color = color;
   }

   public PlayerColor getColor() {
      return color;
   }
}
```

```
package engine;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;
import engine.movement.*;
import engine.piece.*;

import java.util.Vector;

public class Controller implements chess.ChessController {

   private ChessView view ;
   TurnManager turnManager = new TurnManager();
   Piece[][] board = new Piece[8][8];
   private final Vector<Movement> history = new Vector<>();
   // position of both the kings, [0] white king, [1] black king
   private final Position[] kingPosition = {new Position(4, 0), new Position(4, 7)};


   /**
    * Function that populates a part of the chessboard graphically and on the chessboard
    * @param color     the color of a player
    * @param index     the side of the board where we want to put the pieces
    * @param view      the view where we want to add the pieces
    */
   private void loadChess(PlayerColor color, int index, ChessView view){

      board[0][index] = new Rook(PieceType.ROOK, color);
      view.putPiece(board[0][index].getType(), color, 0, index);
      board[1][index] = new Knight(PieceType.KNIGHT, color);
      view.putPiece(board[1][index].getType(), color, 1, index);
      board[2][index] = new Bishop(PieceType.BISHOP, color);
      view.putPiece(board[2][index].getType(), color, 2, index);
      board[3][index] = new Queen(PieceType.QUEEN, color);
      view.putPiece(board[3][index].getType(), color, 3, index);
      board[4][index] = new King(PieceType.KING, color);
      view.putPiece(board[4][index].getType(), color, 4, index);
      board[5][index] = new Bishop(PieceType.BISHOP, color);
      view.putPiece(board[5][index].getType(), color, 5, index);
      board[6][index] = new Knight(PieceType.KNIGHT, color);
      view.putPiece(board[6][index].getType(), color, 6, index);
      board[7][index] = new Rook(PieceType.ROOK, color);
      view.putPiece(board[7][index].getType(), color, 7, index);

      int pawn = (color == PlayerColor.WHITE) ? 1 : 6 ;
      for (int i = 0; i < board.length; ++i) {
         board[i][pawn] = new Pawn(PieceType.PAWN, color);
         view.putPiece(board[i][pawn].getType(), color, i, pawn);
      }
   }

   /**
    * Internal class that represent a movement
    */
   private static class Movement {

      private final PieceType piece;
      private final int x;
      private final int y;
      public Movement(PieceType piece, int x, int y) {
         this.piece = piece;
         this.x = x;
         this.y = y;
```

```java
      }
   }

   /**
    * Internal class that represent a position
    */
   private static class Position{
      private int x;
      private int y;

      public Position(int x, int y) {
         this.x = x;
         this.y = y;
      }

      public void setPosition(int x, int y) {
         this.x = x;
         this.y = y;
      }

   }

   /**
    * Function that initialise the chessboard
    * @param view  view that we want to initialise
    */
   private void init(ChessView view){

      loadChess(PlayerColor.WHITE, 0, view);
      loadChess(PlayerColor.BLACK, 7, view);
   }

   /**
    * Start the logic of the programme
    * @param view la vue à utiliser
    */
   @Override
   public void start(ChessView view) {
      this.view = view;
      view.startView();

   }

   /**
    * Function that graphically moves a piece from one position to another and adds movement to the history
    * @param type      type of piece that we want to move
    * @param color     color of the player moving the piece
    * @param fromX     x coordinate where the piece start
    * @param fromY     y coordinate where the piece start
    * @param toX       x coordinate where the piece will move
    * @param toY       y coordinate where the piece will move
    */
   private void movePiece(PieceType type, PlayerColor color, int fromX, int fromY, int toX, int toY){
      view.removePiece(fromX,fromY);
      view.putPiece(type,color,toX,toY);
      history.add(new Movement(type, toX, toY));
   }

   /**
    * Function that checks whether a piece in a certain starting position can be moved to a certain position
  according to the rules of the game
    * @param fromX     x coordinate where the piece start
    * @param fromY     y coordinate where the piece start
    * @param toX       x coordinate where the piece will move
    * @param toY       y coordinate where the piece will move
```

```
     * @return          true if the piece in the start position can move to the destination, false otherwise
     */
    @Override
    public boolean move(int fromX, int fromY, int toX, int toY) {
        Piece piece = board[fromX][fromY];

        // check if we selected a piece
        if(piece == null)
            return false;

        // check if the piece selected is of the same color as the current player
        if(piece.getColor() != turnManager.playerInTurn.getColor())
            return false;

        if(board[toX][toY] != null){
            // check if we try to move to a square that contains a piece of the same color as the piece moved
            if(piece.getColor() == board[toX][toY].getColor()){
                return false;
            }

        }
        boolean isCastling = false;
        boolean enPassant = false;
        boolean canMove = false;
        switch(piece.getType()){
            case QUEEN:
                canMove = Diagonal.move(fromX, fromY, toX, toY)
                        || Straight.move(fromX, fromY, toX, toY);
                break;
            case ROOK:
                canMove = Straight.move(fromX, fromY, toX, toY);
                break;
            case KNIGHT:
                canMove = Lshape.move(fromX, fromY, toX, toY);
                break;
            case BISHOP:
                canMove = Diagonal.move(fromX, fromY, toX, toY);
                break;
            case KING:
                canMove = Mking.move(fromX, fromY, toX, toY);
                if(canMove){
                    // update the new position of the king
                    kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(toX, toY);
                    break;
                }

                if (piece.isHasMoved()){
                    break;
                }else{
                    // if the king didn't move it can castle only if there is a not moved rook at the edge of the board
                    if(fromX < toX){
                        if(board[7][fromY].getType() == PieceType.ROOK)
                            canMove = !board[7][fromY].isHasMoved();
                    }else{
                        if(board[0][fromY].getType() == PieceType.ROOK)
                            canMove = !board[0][fromY].isHasMoved();
                    }

                }
                canMove &= Castling.move(fromX, fromY, toX, toY);
                isCastling = canMove;
                break;
            case PAWN:
                PlayerColor color = null;
                int direction = 0;
                int place = 0;
                // White can only move forward
                if (piece.getColor() == PlayerColor.WHITE) {
                    if (fromY > toY) {
                        return false;
                    }
                    color = PlayerColor.BLACK;
                    direction = 1;
                    place = 4;      // row (starting from 0) in the board where the en-passant capture can be made by the white
                }
                // Black can only move backward in a sense.
                if (piece.getColor() == PlayerColor.BLACK) {
                    if (fromY < toY) {
                        return false;
                    }
                    color = PlayerColor.WHITE;
                    direction = -1;
                    place = 3;      // row (starting from 0) in the board where the en-passant capture can be made by the black
                }

                if(piece.isHasMoved() && board[toX][toY] == null){
                    canMove = (fromX == toX) && (Math.abs(toY - fromY) == 1);
                } else if(board[toX][toY] == null) {
                    canMove = Mpawn.move(fromX, fromY, toX, toY);
                }

                for(int j = -1; j <= 1; j += 2){
                    try{
                        if(toX == fromX + j && toY == fromY + direction){
                            if(board[fromX + j][fromY + direction] != null           // normal capture
                                    && board[fromX + j][fromY + direction].getColor() == color){
                                canMove = true;
                            }else if(fromY == place
                                    && board[fromX + j][fromY].getColor() == color    // en-passant capture
                                    && checkEnPassant(fromX, fromY, j)){
                                enPassant = true;
                            }
                        }
                    }catch (Exception ignored){}
                }
                break;

        }

        // if the piece follow the rules of movement and there are no obstacle on his path
        if(canMove){
            if(!checkNoObstacle(piece, fromX, fromY, toX, toY))
                return false;

            // we move the piece only on the board to check if his movement created a check
            Piece tempPiece = board[toX][toY];
            board[toX][toY] = board[fromX][fromY];
            board[fromX][fromY] = null;
            // if the piece that we want to move is a king we have to change is position in the table
            if(piece.getType() == PieceType.KING){
                kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(toX, toY);
            }
            // verify if with the new position of the piece, the current player's king would be in check
            if(checkCheck(turnManager.playerInTurn.getColor().ordinal())){
                // otherwise, put the piece in the initial state
                board[fromX][fromY] = board[toX][toY];
                board[toX][toY] = tempPiece;
                if(piece.getType() == PieceType.KING){
```

```java
                    kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(fromX, fromY);
                }
                return false;
            }

            // if the king is castling we move the chosen rook at the right spot
            if(isCastling){
                if(fromX < toX){
                    movePiece(PieceType.ROOK, piece.getColor(), 7, fromY, toX - 1, toY);
                }else{
                    movePiece(PieceType.ROOK, piece.getColor(), 0, fromY, toX + 1, toY);
                }
                kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(toX, toY);
            }

            // if a pawn reached the last line it can be promoted to another piece
            if(isPromoting(piece, toY)){
                view.displayMessage("Which piece do you want to promote into?");
                piece = view.askUser("Promotion","Witch piece do you want to promote into?",
                        new Queen(PieceType.QUEEN, piece.getColor()),
                        new Rook(PieceType.ROOK, piece.getColor()),
                        new Bishop(PieceType.BISHOP, piece.getColor()),
                        new Knight(PieceType.KNIGHT, piece.getColor()));
                board[toX][toY] = piece;
            }

            movePiece(piece.getType(), piece.getColor(),fromX, fromY, toX, toY);
            piece.setHasMoved(true);
            // verify if our movement put the opponent king in check
            if(checkCheck(turnManager.playerNotInTurn.getColor().ordinal())){
                view.displayMessage("Check!");
            }
            turnManager.switchTurn();
            // the capture en-passant is a special moves that eats a pawn that is not in the movement's target square
        }else if(enPassant){
            view.removePiece(toX, fromY);
            board[toX][fromY] = null;
            movePiece(piece.getType(), piece.getColor(),fromX, fromY, toX, toY);
            board[toX][toY] = piece;
            piece.setHasMoved(true);
            if(checkCheck(turnManager.playerNotInTurn.getColor().ordinal())){
                view.displayMessage("Check!");
            }
            turnManager.switchTurn();
        }
        return canMove || enPassant;
    }

    /**
     * Function that verify if a pawn reached the last line
     * @param piece     the piece that has moved
     * @param toY       the row that we want to verify
     * @return          true if the piece is a pawn, and it reached the row 0 for black or the row 7 for white
     */

    private boolean isPromoting(Piece piece, int toY){
        return piece.getType() == PieceType.PAWN && (piece.getColor() == PlayerColor.WHITE ? toY == 7 : toY == 0);
    }

    /**
     * Function that checks whether the last move was by an opponent's pawn that has moved two squares near to the current player's pawn
     * @param fromX     x position of the pawn
     * @param fromY     y position of the pawn
     * @param j         modifier to control the left and right of the pawn
     * @return          true if an en-passant capture can be made
     */
    private boolean checkEnPassant(int fromX, int fromY, int j){
        return board[fromX + j][fromY].getType() == PieceType.PAWN
                && history.lastElement().piece == PieceType.PAWN        // if the last moves was a pawn moving near the actual pawn by 2
                && history.lastElement().x == (fromX + j)
                && history.lastElement().y == fromY;
    }

    /**
     * Function that verifies whether any opponent piece can check the king given in parameter
     * @param kingToCheck       king that we want to verify
     * @return                  true if there is someone that can move to the king position, false otherwise
     */
    private boolean checkCheck(int kingToCheck){

        for (int row = 0; row < board.length; ++row) {
            for(int col = 0; col < board.length; ++col) {
                Piece pieceToCheck = board[row][col];
                if( pieceToCheck != null && pieceToCheck.getColor().ordinal() != kingToCheck){
                    if(checkCanMoveTo(pieceToCheck, row, col, kingPosition[kingToCheck].x, kingPosition[kingToCheck].y)
                            && checkNoObstacle(pieceToCheck, row, col, kingPosition[kingToCheck].x, kingPosition[kingToCheck].y)){
                        return true;
                    }
                }
            }
        }
        return false;
    }

    /**
     * Function that checks whether a piece could move in a given square
     * @param piece     piece to check
     * @param fromX     x coordinate where the piece start
     * @param fromY     y coordinate where the piece start
     * @param toX       x coordinate where the piece will move
     * @param toY       y coordinate where the piece will move
     * @return          true if the selected piece could potentially move to a square
     */
    private boolean checkCanMoveTo(Piece piece, int fromX, int fromY, int toX, int toY){
        switch(piece.getType()){
            case QUEEN:
                return Diagonal.move(fromX, fromY, toX, toY)
                        || Straight.move(fromX, fromY, toX, toY);
            case ROOK:
                return Straight.move(fromX, fromY, toX, toY);
            case KNIGHT:
                return Lshape.move(fromX, fromY, toX, toY);
            case BISHOP:
                return Diagonal.move(fromX, fromY, toX, toY);
            case KING:
                return Mking.move(fromX, fromY, toX, toY);
            case PAWN:
                // White can only move forward
                if (piece.getColor() == PlayerColor.WHITE) {
                    if (fromY > toY) {
                        return false;
                    }
                    for(int j = -1; j <= 1; j += 2){
                        if(toX == fromX + j && toY == fromY + 1){
```

```java
                    return true; //normal capture
                }
            }
        }
        // Black can only move backward in a sense.
        if (piece.getColor() == PlayerColor.BLACK) {
            if (fromY < toY) {
                return false;
            }
            for(int j = -1; j<=1; j+=2){
                if(toX == fromX + j && toY == fromY - 1){
                    return true; //normal capture
                }
            }
        }

    }
    return false;
}

/**
 * Function that checks if there are any obstacles in the way of a piece
 * @param piece   piece to check if there are no obstacle on his path
 * @param fromX   x coordinate where the piece start
 * @param fromY   y coordinate where the piece start
 * @param toX     x coordinate where the piece will move
 * @param toY     y coordinate where the piece will move
 * @return        true if there are no obstacles on the path
 */
private boolean checkNoObstacle(Piece piece, int fromX, int fromY, int toX, int toY) {

    boolean noObstacle = true;
    int modX, modY;
    switch (piece.getType()){
        case BISHOP:
        {
            if (fromX < toX && fromY < toY) {
                modX = modY = 1;
            } else if (fromX > toX && fromY > toY) {
                modX = modY = -1;
            } else if (fromX < toX && fromY > toY) {
                modX = 1;
                modY = -1;
            } else {
                modX = -1;
                modY = 1;
            }
            int i = fromX + modX;
            int j = fromY + modY;
            for (; i != toX; i += modX, j += modY) {
                noObstacle &= (board[i][j] == null);
            }
            break;
        }
        case KING:
        case ROOK:
            if(fromX == toX){
                modY = fromY < toY ? 1 : -1;
                for(int j = fromY + modY; j != toY; j += modY){
                    noObstacle &= (board[fromX][j] == null);
                }
            }else{
                modX = fromX < toX ? 1 : -1;
                for(int i = fromX + modX; i != toX; i += modX){
                    noObstacle &= (board[i][fromY] == null);
                    }
                }
                break;
            case QUEEN:
                if(fromX == toX){
                    modY = fromY < toY ? 1 : -1;

                    for(int j = fromY + modY; j != toY; j += modY){
                        noObstacle &= (board[fromX][j] == null);
                    }
                }else if(fromY == toY){
                    modX = fromX < toX ? 1 : -1;

                    for(int i = fromX + modX; i != toX; i += modX){
                        noObstacle &= (board[i][fromY] == null);
                    }
                } else {
                    if (fromX < toX && fromY < toY) {
                        modX = modY = 1;
                    } else if (fromX > toX && fromY > toY) {
                        modX = modY = -1;
                    } else if (fromX < toX) {
                        modX = 1;
                        modY = -1;
                    } else {
                        modX = -1;
                        modY = 1;
                    }
                    int i = fromX + modX;
                    int j = fromY + modY;
                    for (; i != toX; i += modX, j += modY) {
                        noObstacle &= (board[i][j] == null);
                    }
                }
                break;
            case PAWN:
                if (piece.isHasMoved())
                    break;
                switch (piece.getColor()) {
                    case WHITE:
                        if (fromY + 2 == toY) {
                            noObstacle = (board[fromX][fromY + 1] == null);
                        }
                        break;
                    case BLACK:
                        if (fromY - 2 == toY) {
                            noObstacle = (board[fromX][fromY - 1] == null);
                        }
                        break;
                }
        }
        return noObstacle;
    }

    /**
     * Function that starts a new game
     */
    @Override
    public void newGame() {

        for(int i = 0; i < board.length ; ++i){
            for(int j = 0; j < board[0].length; ++j){
                view.removePiece(i,j);
                board[i][j] = null;
            }
        }
        init(view);
        turnManager.resetPlayerInTurn();
    }
}
```

```java
package engine;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.PlayerColor;

public class TurnManager {

    Player playerInTurn;
    Player playerNotInTurn;
    Player blackPlayer;
    Player whitePlayer;

    public TurnManager()
    {
        playerInTurn = new Player(PlayerColor.WHITE);
        whitePlayer = playerInTurn;
        playerNotInTurn = new Player(PlayerColor.BLACK);
        blackPlayer = playerNotInTurn;
    }

    public void switchTurn()
    {
        Player temp  = playerInTurn;
        playerInTurn = playerNotInTurn;
        playerNotInTurn = temp;
    }

    public void resetPlayerInTurn() {
        playerInTurn = whitePlayer;
        playerNotInTurn = blackPlayer;
    }
}
```

```java
package engine.piece;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.PieceType;
import chess.PlayerColor;

public class King extends Piece {

   private boolean hasMoved;

   public King(PieceType type, PlayerColor color) {
      super(type, color);
      hasMoved = false;
   }

   @Override
   public boolean isHasMoved() {
      return hasMoved;
   }

   @Override
   public void setHasMoved(boolean hasMoved) {
      this.hasMoved = hasMoved;
   }

}

```

```java
package engine.piece;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.PieceType;
import chess.PlayerColor;

public class Pawn extends Piece {

   private boolean hasMoved;

   public Pawn(PieceType type, PlayerColor color) {
      super(type,color);
      hasMoved = false;
   }

   @Override
   public boolean isHasMoved() {
      return hasMoved;
   }

   @Override
   public void setHasMoved(boolean hasMoved) {
      this.hasMoved = hasMoved;
   }

}

```

```java
package engine.piece;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;

public class Rook extends Piece implements ChessView.UserChoice {

   boolean hasMoved;

   public Rook(PieceType type, PlayerColor color) {
      super(type, color);
      hasMoved = false;
   }

   @Override
   public boolean isHasMoved() {
      return hasMoved;
   }

   @Override
   public void setHasMoved(boolean hasMoved) {
      this.hasMoved = hasMoved;
   }

   @Override
   public String textValue() {
      return "Rook";
   }

   @Override
   public String toString() {
      return textValue();
   }

}
```

```java
package engine.piece;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.PieceType;
import chess.PlayerColor;

public abstract class Piece {

    private final PieceType type;
    private final PlayerColor color;
    private boolean hasMoved;

    public Piece(PieceType type, PlayerColor color) {
        this.type = type;
        this.color = color;
        hasMoved = false;
    }

    public PieceType getType() {
        return type;
    }

    public PlayerColor getColor() {
        return color;
    }

    public boolean isHasMoved() {
        return hasMoved;
    }

    public void setHasMoved(boolean hasMoved) {
        this.hasMoved = hasMoved;
    }
}

```

```java
package engine.piece;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;

public class Queen extends Piece implements ChessView.UserChoice {

   public Queen(PieceType type, PlayerColor color) {
      super(type, color);

   }

   @Override
   public String textValue() {
      return "Queen";
   }

   @Override
   public String toString() {
      return textValue();
   }
}

```

```java
package engine.piece;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;

public class Bishop extends Piece implements ChessView.UserChoice {

   public Bishop(PieceType type, PlayerColor color) {
      super(type, color);
   }

   @Override
   public String textValue() {
      return "Bishop";
   }

   @Override
   public String toString() {
      return textValue();
   }
}

```

```java
package engine.piece;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;

public class Knight extends Piece implements ChessView.UserChoice {

   public Knight(PieceType type, PlayerColor color) {
      super(type, color);
   }


   @Override
   public String textValue() {
      return "Knight";
   }

   @Override
   public String toString() {
      return textValue();
   }
}

```

```java
package engine.movement;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

public interface Move {
   static boolean move(int fromX, int fromY, int toX, int toY){
      return false;
   }
}

```

```java
package engine.movement;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

public class Mking implements Move {

   public static Boolean move(int fromX, int fromY, int toX, int toY) {
      return Math.abs(fromX - toX) < 2 && Math.abs(fromY - toY) < 2;
   }
}

```

```java
package engine.movement;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

public class Mpawn implements Move {

    public static Boolean move(int fromX, int fromY, int toX, int toY) {
        return (fromX == toX) && (Math.abs(toY - fromY) < 3);
    }
}

```

```java
package engine.movement;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

public class Lshape implements Move {

    public static Boolean move(int fromX, int fromY, int toX, int toY) {
        int x = Math.abs(fromX - toX);
        int y = Math.abs(fromY - toY);
        return x * y == 2;
    }
}
```

```java
package engine.movement;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

public class Castling implements Move {

   public static Boolean move(int fromX, int fromY, int toX, int toY) {
      return fromY == toY && (Math.abs(fromX - toX) == 2);
   }
}
```

```java
package engine.movement;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

public class Diagonal implements Move {

   public static boolean move(int fromX, int fromY, int toX, int toY) {
      return Math.abs(fromX -  toX) == Math.abs(fromY - toY);
   }
}
```

```java
package engine.movement;
/* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */

public class Straight implements Move {

   public static Boolean move(int fromX, int fromY, int toX, int toY) {
      return fromX == toX || fromY == toY;
   }
}
```