

```
1 package engine;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.PlayerColor;
5
6
7 public class Player {
8     private final PlayerColor color;
9
10    public Player(PlayerColor color){
11        this.color = color;
12    }
13
14    public PlayerColor getColor() {
15        return color;
16    }
17 }
```

```

1  package engine;
2  /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4  import chess.ChessView;
5  import chess.PieceType;
6  import chess.PlayerColor;
7  import engine.movement.*;
8  import engine.piece.*;
9
10 import java.util.Vector;
11
12 public class Controller implements chess.ChessController {
13
14     private ChessView view ;
15     TurnManager turnManager = new TurnManager();
16     Piece[][] board = new Piece[8][8];
17     private final Vector<Movement> history = new Vector<>();
18     // position of both the kings, [0] white king, [1] black king
19     private final Position[] kingPosition = {new Position(4, 0), new Position(4, 7)};
20
21
22     /**
23      * Function that populates a part of the chessboard graphically and on the chessboard
24      * @param color    the color of a player
25      * @param index    the side of the board where we want to put the pieces
26      * @param view     the view where we want to add the pieces
27      */
28     private void loadChess(PlayerColor color, int index, ChessView view){
29
30         board[0][index] = new Rook(PieceType.ROOK, color);
31         view.putPiece(board[0][index].getType(), color, 0, index);
32         board[1][index] = new Knight(PieceType.KNIGHT, color);
33         view.putPiece(board[1][index].getType(), color, 1, index);
34         board[2][index] = new Bishop(PieceType.BISHOP, color);
35         view.putPiece(board[2][index].getType(), color, 2, index);
36         board[3][index] = new Queen(PieceType.QUEEN, color);
37         view.putPiece(board[3][index].getType(), color, 3, index);
38         board[4][index] = new King(PieceType.KING, color);
39         view.putPiece(board[4][index].getType(), color, 4, index);
40         board[5][index] = new Bishop(PieceType.BISHOP, color);
41         view.putPiece(board[5][index].getType(), color, 5, index);
42         board[6][index] = new Knight(PieceType.KNIGHT, color);
43         view.putPiece(board[6][index].getType(), color, 6, index);
44         board[7][index] = new Rook(PieceType.ROOK, color);
45         view.putPiece(board[7][index].getType(), color, 7, index);
46
47         int pawn = (color == PlayerColor.WHITE) ? 1 : 6 ;
48         for (int i = 0; i < board.length; ++i) {
49             board[i][pawn] = new Pawn(PieceType.PAWN, color);
50             view.putPiece(board[i][pawn].getType(), color, i, pawn);
51         }
52     }
53
54     /**
55      * Internal class that represent a movement
56      */
57     private static class Movement {
58
59         private final PieceType piece;
60         private final int x;
61         private final int y;
62         public Movement(PieceType piece, int x, int y) {
63             this.piece = piece;
64             this.x = x;
65             this.y = y;

```

```

66     }
67 }
68
69 /**
70  * Internal class that represent a position
71  */
72 private static class Position{
73     private int x;
74     private int y;
75
76     public Position(int x, int y) {
77         this.x = x;
78         this.y = y;
79     }
80
81     public void setPosition(int x, int y) {
82         this.x = x;
83         this.y = y;
84     }
85
86 }
87
88 /**
89  * Function that initialise the chessboard
90  * @param view view that we want to initialise
91  */
92 private void init(ChessView view){
93
94     loadChess(PlayerColor.WHITE, 0, view);
95     loadChess(PlayerColor.BLACK, 7, view);
96 }
97
98 /**
99  * Start the logic of the programme
100  * @param view la vue à utiliser
101  */
102 @Override
103 public void start(ChessView view) {
104     this.view = view;
105     view.startView();
106
107 }
108
109 /**
110  * Function that graphically moves a piece from one position to another and adds movement to the history
111  * @param type type of piece that we want to move
112  * @param color color of the player moving the piece
113  * @param fromX x coordinate where the piece start
114  * @param fromY y coordinate where the piece start
115  * @param toX x coordinate where the piece will move
116  * @param toY y coordinate where the piece will move
117  */
118 private void movePiece(PieceType type, PlayerColor color, int fromX, int fromY, int toX, int toY){
119     view.removePiece(fromX,fromY);
120     view.putPiece(type,color,toX,toY);
121     history.add(new Movement(type, toX, toY));
122 }
123
124 /**
125  * 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
126  * @param fromX x coordinate where the piece start
127  * @param fromY y coordinate where the piece start
128  * @param toX x coordinate where the piece will move
129  * @param toY y coordinate where the piece will move

```

```

130  * @return      true if the piece in the start position can move to the destination, false otherwise
131  */
132  @Override
133  public boolean move(int fromX, int fromY, int toX, int toY) {
134      Piece piece = board[fromX][fromY];
135
136      // check if we selected a piece
137      if(piece == null)
138          return false;
139
140      // check if the piece selected is of the same color as the current player
141      if(piece.getColor() != turnManager.playerInTurn.getColor())
142          return false;
143
144      if(board[toX][toY] != null){
145          // check if we try to move to a square that contains a piece of the same color as the piece moved
146          if(piece.getColor() == board[toX][toY].getColor()){
147              return false;
148          }
149      }
150  }
151  boolean isCastling = false;
152  boolean enPassant = false;
153  boolean canMove = false;
154  switch(piece.getType()){
155      case QUEEN:
156          canMove = Diagonal.move(fromX, fromY, toX, toY)
157              || Straight.move(fromX, fromY, toX, toY);
158          break;
159      case ROOK:
160          canMove = Straight.move(fromX, fromY, toX, toY);
161          break;
162      case KNIGHT:
163          canMove = Lshape.move(fromX, fromY, toX, toY);
164          break;
165      case BISHOP:
166          canMove = Diagonal.move(fromX, fromY, toX, toY);
167          break;
168      case KING:
169          canMove = Mking.move(fromX, fromY, toX, toY);
170          if(canMove){
171              // update the new position of the king
172              kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(toX, toY);
173              break;
174          }
175
176          if (piece.isHasMoved()){
177              break;
178          }else{
179              // if the king didn't move it can castle only if there is a not moved rook at the edge of the board
180              if(fromX < toX){
181                  if(board[7][fromY].getType() == PieceType.ROOK)
182                      canMove = !board[7][fromY].isHasMoved();
183              }else{
184                  if(board[0][fromY].getType() == PieceType.ROOK)
185                      canMove = !board[0][fromY].isHasMoved();
186              }
187          }
188
189          canMove &= Castling.move(fromX, fromY, toX, toY);
190          isCastling = canMove;
191          break;
192      case PAWN:
193          PlayerColor color = null;
194          int direction = 0;

```

```

195     int place = 0;
196     // White can only move forward
197     if (piece.getColor() == PlayerColor.WHITE) {
198         if (fromY > toY) {
199             return false;
200         }
201         color = PlayerColor.BLACK;
202         direction = 1;
203         place = 4;    // row (starting from 0) in the board where the en-passant capture can be made by
the white
204     }
205     // Black can only move backward in a sense.
206     if (piece.getColor() == PlayerColor.BLACK) {
207         if (fromY < toY) {
208             return false;
209         }
210         color = PlayerColor.WHITE;
211         direction = -1;
212         place = 3;    // row (starting from 0) in the board where the en-passant capture can be made by
the black
213     }
214
215     if(piece.isHasMoved() && board[toX][toY] == null){
216         canMove = (fromX == toX) && (Math.abs(toY - fromY) == 1);
217     } else if(board[toX][toY] == null) {
218         canMove = Mpawn.move(fromX, fromY, toX, toY);
219     }
220
221     for(int j = -1; j <= 1; j += 2){
222         try{
223             if(toX == fromX + j && toY == fromY + direction){
224                 if(board[fromX + j][fromY + direction] != null    // normal capture
225                     && board[fromX + j][fromY + direction].getColor() == color){
226                     canMove = true;
227                 }else if(fromY == place
228                     && board[fromX + j][fromY].getColor() == color    // en-passant capture
229                     && checkEnPassant(fromX, fromY, j)){
230                     enPassant = true;
231                 }
232             }
233         }catch (Exception ignored){}
234     }
235     break;
236 }
237
238
239 // if the piece follow the rules of movement and there are no obstacle on his path
240 if(canMove){
241     if(!checkNoObstacle(piece, fromX, fromY, toX, toY))
242         return false;
243
244     // we move the piece only on the board to check if his movement created a check
245     Piece tempPiece = board[toX][toY];
246     board[toX][toY] = board[fromX][fromY];
247     board[fromX][fromY] = null;
248     // if the piece that we want to move is a king we have to change is position in the table
249     if(piece.getType() == PieceType.KING){
250         kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(toX, toY);
251     }
252     // verify if with the new position of the piece, the current player's king would be in check
253     if(checkCheck(turnManager.playerInTurn.getColor().ordinal())){
254         // otherwise, put the piece in the initial state
255         board[fromX][fromY] = board[toX][toY];
256         board[toX][toY] = tempPiece;
257         if(piece.getType() == PieceType.KING){

```

```

258         kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(fromX, fromY);
259     }
260     return false;
261 }
262
263 // if the king is castling we move the chosen rook at the right spot
264 if(isCastling){
265     if(fromX < toX){
266         movePiece(PieceType.ROOK, piece.getColor(), 7, fromY, toX - 1, toY);
267     }else{
268         movePiece(PieceType.ROOK, piece.getColor(), 0, fromY, toX + 1, toY);
269     }
270     kingPosition[piece.getColor() == PlayerColor.WHITE ? 0 : 1].setPosition(toX, toY);
271 }
272
273 // if a pawn reached the last line it can be promoted to another piece
274 if(isPromoting(piece, toY)){
275     view.displayMessage("Which piece do you want to promote into?");
276     piece = view.askUser("Promotion", "Witch piece do you want to promote into?",
277         new Queen(PieceType.QUEEN, piece.getColor()),
278         new Rook(PieceType.ROOK, piece.getColor()),
279         new Bishop(PieceType.BISHOP, piece.getColor()),
280         new Knight(PieceType.KNIGHT, piece.getColor()));
281     board[toX][toY] = piece;
282 }
283
284 movePiece(piece.getType(), piece.getColor(), fromX, fromY, toX, toY);
285 piece.setHasMoved(true);
286 // verify if our movement put the opponent king in check
287 if(checkCheck(turnManager.playerNotInTurn.getColor().ordinal())){
288     view.displayMessage("Check!");
289 }
290 turnManager.switchTurn();
291 // the capture en-passant is a special moves that eats a pawn that is not in the movement's target
square
292 }else if(enPassant){
293     view.removePiece(toX, fromY);
294     board[toX][fromY] = null;
295     movePiece(piece.getType(), piece.getColor(), fromX, fromY, toX, toY);
296     board[toX][toY] = piece;
297     piece.setHasMoved(true);
298     if(checkCheck(turnManager.playerNotInTurn.getColor().ordinal())){
299         view.displayMessage("Check!");
300     }
301     turnManager.switchTurn();
302 }
303 return canMove || enPassant;
304 }
305
306 /**
307  * Function that verify if a pawn reached the last line
308  * @param piece the piece that has moved
309  * @param toY the row that we want to verify
310  * @return true if the piece is a pawn, and it reached the row 0 for black or the row 7 for white
311  */
312
313 private boolean isPromoting(Piece piece, int toY){
314     return piece.getType() == PieceType.PAWN && (piece.getColor() == PlayerColor.WHITE ? toY == 7 : toY
== 0);
315 }
316
317 /**
318  * 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
319  * @param fromX x position of the pawn

```

```

320  * @param fromY    y position of the pawn
321  * @param j        modifier to control the left and right of the pawn
322  * @return         true if an en-passant capture can be made
323  */
324  private boolean checkEnPassant(int fromX, int fromY, int j){
325      return board[fromX + j][fromY].getType() == PieceType.PAWN
326      && history.lastElement().piece == PieceType.PAWN    // if the last moves was a pawn moving near
the actual pawn by 2
327      && history.lastElement().x == (fromX + j)
328      && history.lastElement().y == fromY;
329  }
330
331  /**
332   * Function that verifies whether any opponent piece can check the king given in parameter
333   * @param kingToCheck    king that we want to verify
334   * @return               true if there is someone that can move to the king position, false otherwise
335   */
336  private boolean checkCheck(int kingToCheck){
337
338      for (int row = 0; row < board.length; ++row) {
339          for(int col = 0; col < board.length; ++col) {
340              Piece pieceToCheck = board[row][col];
341              if( pieceToCheck != null && pieceToCheck.getColor().ordinal() != kingToCheck){
342                  if(checkCanMoveTo(pieceToCheck, row, col, kingPosition[kingToCheck].x, kingPosition[
kingToCheck].y)
343                  && checkNoObstacle(pieceToCheck, row, col, kingPosition[kingToCheck].x, kingPosition[
kingToCheck].y)){
344                      return true;
345                  }
346              }
347          }
348      }
349      return false;
350  }
351
352  /**
353   * Function that checks whether a piece could move in a given square
354   * @param piece    piece to check
355   * @param fromX    x coordinate where the piece start
356   * @param fromY    y coordinate where the piece start
357   * @param toX      x coordinate where the piece will move
358   * @param toY      y coordinate where the piece will move
359   * @return         true if the selected piece could potentially move to a square
360   */
361  private boolean checkCanMoveTo(Piece piece, int fromX, int fromY, int toX, int toY){
362      switch(piece.getType()){
363          case QUEEN:
364              return Diagonal.move(fromX, fromY, toX, toY)
|| Straight.move(fromX, fromY, toX, toY);
365          case ROOK:
366              return Straight.move(fromX, fromY, toX, toY);
367          case KNIGHT:
368              return Lshape.move(fromX, fromY, toX, toY);
369          case BISHOP:
370              return Diagonal.move(fromX, fromY, toX, toY);
371          case KING:
372              return Mking.move(fromX, fromY, toX, toY);
373          case PAWN:
374              // White can only move forward
375              if (piece.getColor() == PlayerColor.WHITE) {
376                  if (fromY > toY) {
377                      return false;
378                  }
379              }
380              for(int j = -1; j <= 1; j += 2){
381                  if(toX == fromX + j && toY == fromY + 1){

```

```

382         return true; //normal capture
383     }
384 }
385 }
386 // Black can only move backward in a sense.
387 if (piece.getColor() == PlayerColor.BLACK) {
388     if (fromY < toY) {
389         return false;
390     }
391     for(int j = -1; j<=1; j+=2){
392         if(toX == fromX + j && toY == fromY - 1){
393             return true; //normal capture
394         }
395     }
396 }
397
398 }
399 return false;
400 }
401
402 /**
403  * Function that checks if there are any obstacles in the way of a piece
404  * @param piece piece to check if there are no obstacle on his path
405  * @param fromX x coordinate where the piece start
406  * @param fromY y coordinate where the piece start
407  * @param toX x coordinate where the piece will move
408  * @param toY y coordinate where the piece will move
409  * @return true if there are no obstacles on the path
410  */
411 private boolean checkNoObstacle(Piece piece, int fromX, int fromY, int toX, int toY) {
412
413     boolean noObstacle = true;
414     int modX, modY;
415     switch (piece.getType()){
416         case BISHOP:
417         {
418             if (fromX < toX && fromY < toY) {
419                 modX = modY = 1;
420             } else if (fromX > toX && fromY > toY) {
421                 modX = modY = -1;
422             } else if (fromX < toX && fromY > toY) {
423                 modX = 1;
424                 modY = -1;
425             } else {
426                 modX = -1;
427                 modY = 1;
428             }
429             int i = fromX + modX;
430             int j = fromY + modY;
431             for (; i != toX; i += modX, j += modY) {
432                 noObstacle &= (board[i][j] == null);
433             }
434             break;
435         }
436         case KING:
437         case ROOK:
438             if(fromX == toX){
439                 modY = fromY < toY ? 1 : -1;
440                 for(int j = fromY + modY; j != toY; j += modY){
441                     noObstacle &= (board[fromX][j] == null);
442                 }
443             }else{
444                 modX = fromX < toX ? 1 : -1;
445                 for(int i = fromX + modX; i != toX; i += modX){
446                     noObstacle &= (board[i][fromY] == null);

```



```

447     }
448 }
449 break;
450 case QUEEN:
451     if(fromX == toX){
452         modY = fromY < toY ? 1 : -1;
453
454         for(int j = fromY + modY; j != toY; j += modY){
455             noObstacle &= (board[fromX][j] == null);
456         }
457     } else if(fromY == toY){
458         modX = fromX < toX ? 1 : -1;
459
460         for(int i = fromX + modX; i != toX; i += modX){
461             noObstacle &= (board[i][fromY] == null);
462         }
463     } else {
464         if (fromX < toX && fromY < toY) {
465             modX = modY = 1;
466         } else if (fromX > toX && fromY > toY) {
467             modX = modY = -1;
468         } else if (fromX < toX) {
469             modX = 1;
470             modY = -1;
471         } else {
472             modX = -1;
473             modY = 1;
474         }
475         int i = fromX + modX;
476         int j = fromY + modY;
477         for (; i != toX; i += modX, j += modY) {
478             noObstacle &= (board[i][j] == null);
479         }
480     }
481     break;
482 case PAWN:
483     if (piece.isHasMoved())
484         break;
485     switch (piece.getColor()) {
486     case WHITE:
487         if (fromY + 2 == toY) {
488             noObstacle = (board[fromX][fromY + 1] == null);
489         }
490         break;
491     case BLACK:
492         if (fromY - 2 == toY) {
493             noObstacle = (board[fromX][fromY - 1] == null);
494         }
495         break;
496     }
497 }
498 return noObstacle;
499 }
500
501 /**
502  * Function that starts a new game
503  */
504 @Override
505 public void newGame() {
506
507     for(int i = 0; i < board.length ; ++i){
508         for(int j = 0; j < board[0].length; ++j){
509             view.removePiece(i,j);
510             board[i][j] = null;
511         }

```

```
512     }  
513     init(view);  
514     turnManager.resetPlayerInTurn();  
515 }  
516 }
```

```
1 package engine;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.PlayerColor;
5
6 public class TurnManager {
7
8     Player playerInTurn;
9     Player playerNotInTurn;
10    Player blackPlayer;
11    Player whitePlayer;
12
13    public TurnManager()
14    {
15        playerInTurn = new Player(PlayerColor.WHITE);
16        whitePlayer = playerInTurn;
17        playerNotInTurn = new Player(PlayerColor.BLACK);
18        blackPlayer = playerNotInTurn;
19    }
20
21    public void switchTurn()
22    {
23        Player temp = playerInTurn;
24        playerInTurn = playerNotInTurn;
25        playerNotInTurn = temp;
26    }
27
28    public void resetPlayerInTurn() {
29        playerInTurn = whitePlayer;
30        playerNotInTurn = blackPlayer;
31    }
32 }
```

```
1 package engine.piece;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.PieceType;
5 import chess.PlayerColor;
6
7 public class King extends Piece {
8
9     private boolean hasMoved;
10
11     public King(PieceType type, PlayerColor color) {
12         super(type, color);
13         hasMoved = false;
14     }
15
16     @Override
17     public boolean isHasMoved() {
18         return hasMoved;
19     }
20
21     @Override
22     public void setHasMoved(boolean hasMoved) {
23         this.hasMoved = hasMoved;
24     }
25
26 }
27
```

```
1 package engine.piece;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.PieceType;
5 import chess.PlayerColor;
6
7 public class Pawn extends Piece {
8
9     private boolean hasMoved;
10
11     public Pawn(PieceType type, PlayerColor color) {
12         super(type,color);
13         hasMoved = false;
14     }
15
16     @Override
17     public boolean isHasMoved() {
18         return hasMoved;
19     }
20
21     @Override
22     public void setHasMoved(boolean hasMoved) {
23         this.hasMoved = hasMoved;
24     }
25
26 }
27
```

```
1 package engine.piece;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.ChessView;
5 import chess.PieceType;
6 import chess.PlayerColor;
7
8 public class Rook extends Piece implements ChessView.UserChoice {
9
10     boolean hasMoved;
11
12     public Rook(PieceType type, PlayerColor color) {
13         super(type, color);
14         hasMoved = false;
15     }
16
17     @Override
18     public boolean isHasMoved() {
19         return hasMoved;
20     }
21
22     @Override
23     public void setHasMoved(boolean hasMoved) {
24         this.hasMoved = hasMoved;
25     }
26
27     @Override
28     public String textValue() {
29         return "Rook";
30     }
31
32     @Override
33     public String toString() {
34         return textValue();
35     }
36
37 }
38
```

```
1 package engine.piece;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.PieceType;
5 import chess.PlayerColor;
6
7 public abstract class Piece {
8
9     private final PieceType type;
10    private final PlayerColor color;
11    private boolean hasMoved;
12
13    public Piece(PieceType type, PlayerColor color) {
14        this.type = type;
15        this.color = color;
16        hasMoved = false;
17    }
18
19    public PieceType getType() {
20        return type;
21    }
22
23    public PlayerColor getColor() {
24        return color;
25    }
26
27    public boolean isHasMoved() {
28        return hasMoved;
29    }
30
31    public void setHasMoved(boolean hasMoved) {
32        this.hasMoved = hasMoved;
33    }
34 }
35
```

```
1 package engine.piece;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.ChessView;
5 import chess.PieceType;
6 import chess.PlayerColor;
7
8 public class Queen extends Piece implements ChessView.UserChoice {
9
10     public Queen(PieceType type, PlayerColor color) {
11         super(type, color);
12     }
13
14     @Override
15     public String textValue() {
16         return "Queen";
17     }
18
19     @Override
20     public String toString() {
21         return textValue();
22     }
23 }
24 }
25
```



```
1 package engine.piece;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.ChessView;
5 import chess.PieceType;
6 import chess.PlayerColor;
7
8 public class Bishop extends Piece implements ChessView.UserChoice {
9
10     public Bishop(PieceType type, PlayerColor color) {
11         super(type, color);
12     }
13
14     @Override
15     public String textValue() {
16         return "Bishop";
17     }
18
19     @Override
20     public String toString() {
21         return textValue();
22     }
23 }
24
```

```
1 package engine.piece;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 import chess.ChessView;
5 import chess.PieceType;
6 import chess.PlayerColor;
7
8 public class Knight extends Piece implements ChessView.UserChoice {
9
10     public Knight(PieceType type, PlayerColor color) {
11         super(type, color);
12     }
13
14
15     @Override
16     public String textValue() {
17         return "Knight";
18     }
19
20     @Override
21     public String toString() {
22         return textValue();
23     }
24 }
25
```

```
1 package engine.movement;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 public interface Move {
5     static boolean move(int fromX, int fromY, int toX, int toY){
6         return false;
7     }
8 }
9
```

```
1 package engine.movement;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 public class Mking implements Move {
5
6     public static Boolean move(int fromX, int fromY, int toX, int toY) {
7         return Math.abs(fromX - toX) < 2 && Math.abs(fromY - toY) < 2;
8     }
9 }
10
```

```
1 package engine.movement;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 public class Mpawn implements Move {
5
6     public static Boolean move(int fromX, int fromY, int toX, int toY) {
7         return (fromX == toX) && (Math.abs(toY - fromY) < 3);
8     }
9 }
10
```

```
1 package engine.movement;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 public class Lshape implements Move {
5
6     public static Boolean move(int fromX, int fromY, int toX, int toY) {
7         int x = Math.abs(fromX - toX);
8         int y = Math.abs(fromY - toY);
9         return x * y == 2;
10    }
11 }
```

```
1 package engine.movement;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 public class Castling implements Move {
5
6     public static Boolean move(int fromX, int fromY, int toX, int toY) {
7         return fromY == toY && (Math.abs(fromX - toX) == 2);
8     }
9 }
```

```
1 package engine.movement;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 public class Diagonal implements Move {
5
6     public static boolean move(int fromX, int fromY, int toX, int toY) {
7         return Math.abs(fromX - toX) == Math.abs(fromY - toY);
8     }
9 }
10
```



```
1 package engine.movement;
2 /* Auteurs: Akoumba Erica Ludivine, Pontarolo Stefano */
3
4 public class Straight implements Move {
5
6     public static Boolean move(int fromX, int fromY, int toX, int toY) {
7         return fromX == toX || fromY == toY;
8     }
9 }
```