A Game for Kings

AND COMPUTING STUDENTS, PROJECT MANUAL PART 1

Getting Started with the Chess Project

As discussed previously you are required to create a chess game for the AI continuous assessments. You will find on your Moodle page an archived file to help you get started with the project (**this is not a NetBeans project file!**). You do **not** need to be a Chess player to be able to complete the assessments. You do need to know the rules of chess. This practical book will guide you through part 1 that enables you to build the architecture of the Chess game. But first, before getting started research and summarise three AI strategies that have been used in developing solutions for Chess, clearly describing the theory and concepts that underpin the AI techniques identified (Submission week two on Moodle (**CA 1**)).

Getting started, you should compile the code and check out what is working. After compiling and running the ChessProject file, the following interface as seen in Figure 1 should appear.

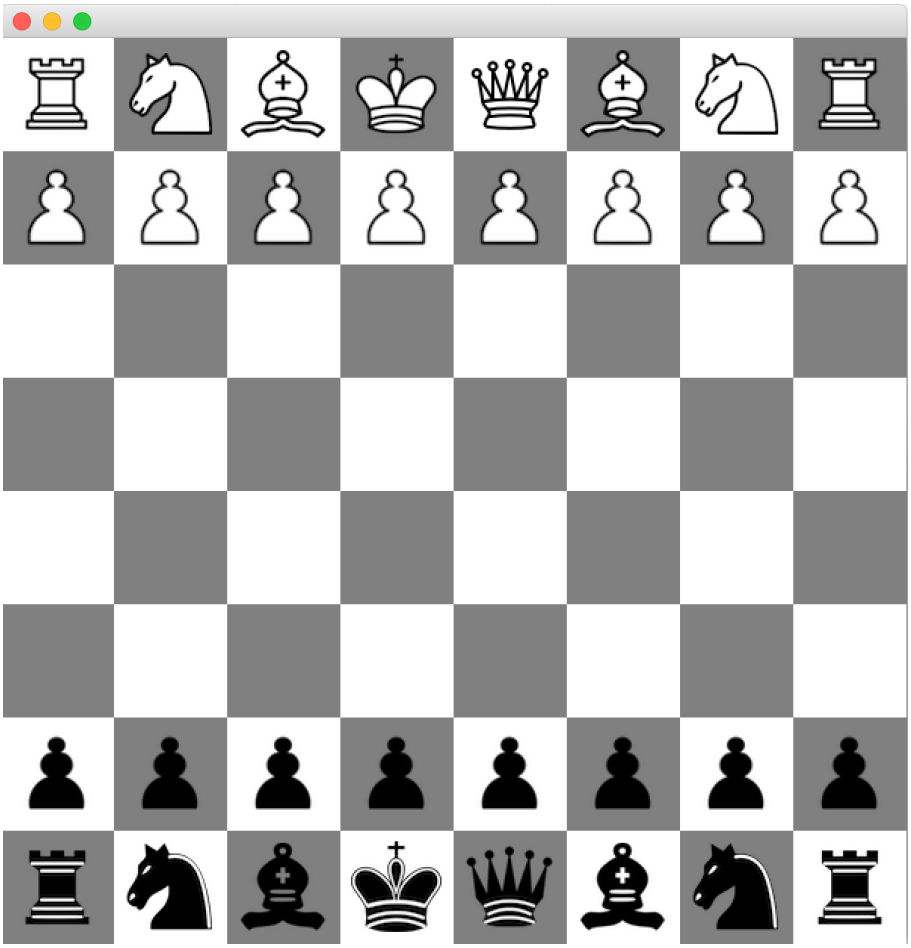

Figure 1: The basic interface for the Chess game

Pawn Movements

So a Pawn is able to move either two or one squares forward on its first move, but only one square after that. The Pawn is the only piece that cannot move backwards in chess...so be careful when committing a pawn forward. A Pawn is able to take any of the opponent's pieces but they have to be one square forward and one square over, i.e. in a diagonal direction from the Pawns original position. If a Pawn makes it to the top of the opponent's side, the Pawn can turn into any other piece, for demonstration purposes the Pawn in the sample code provided turns into a Queen.

The only piece that is able to move at the moment is the White Pawn, which can do the following:

- Move one / two squares on its first move in a straight line.
- Only move one square at a time after making the initial move.
- If the pawn makes it to the top of the board it becomes a queen.
- A pawn can take an enemy piece by moving one square in a diagonal movement.

There are at least two problems (actually three problems) with the code provided...can you find out what is wrong with the pawn movements and fix the problems. Below you will find some interfaces to help you identify the problems that you need to complete.

Hint 1: Consider the following interface, does this look like a feasible situation?

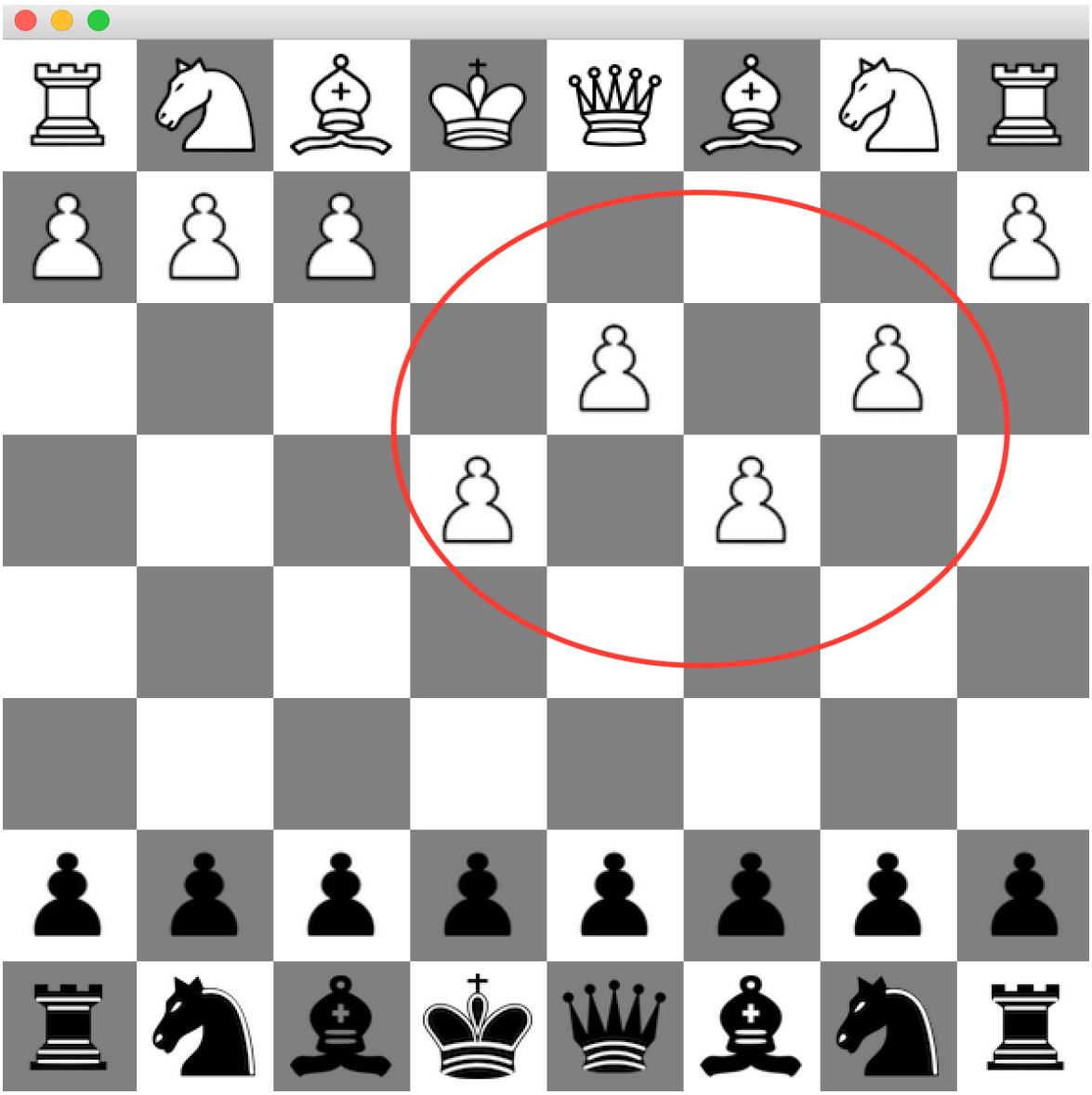

Figure 2: First issue with the White Pawn movements

Chess is a two player game and you need to restrict the movements to make sure that each colour gets only one move per turn. I would **recommend** not implementing this until you get all the pieces moving first or perhaps implementing it and also adding in some code to allow you run in a debug mode.

Hint 2: Consider the following interface, should the highlighted white pawn be able to take the highlighted black piece?

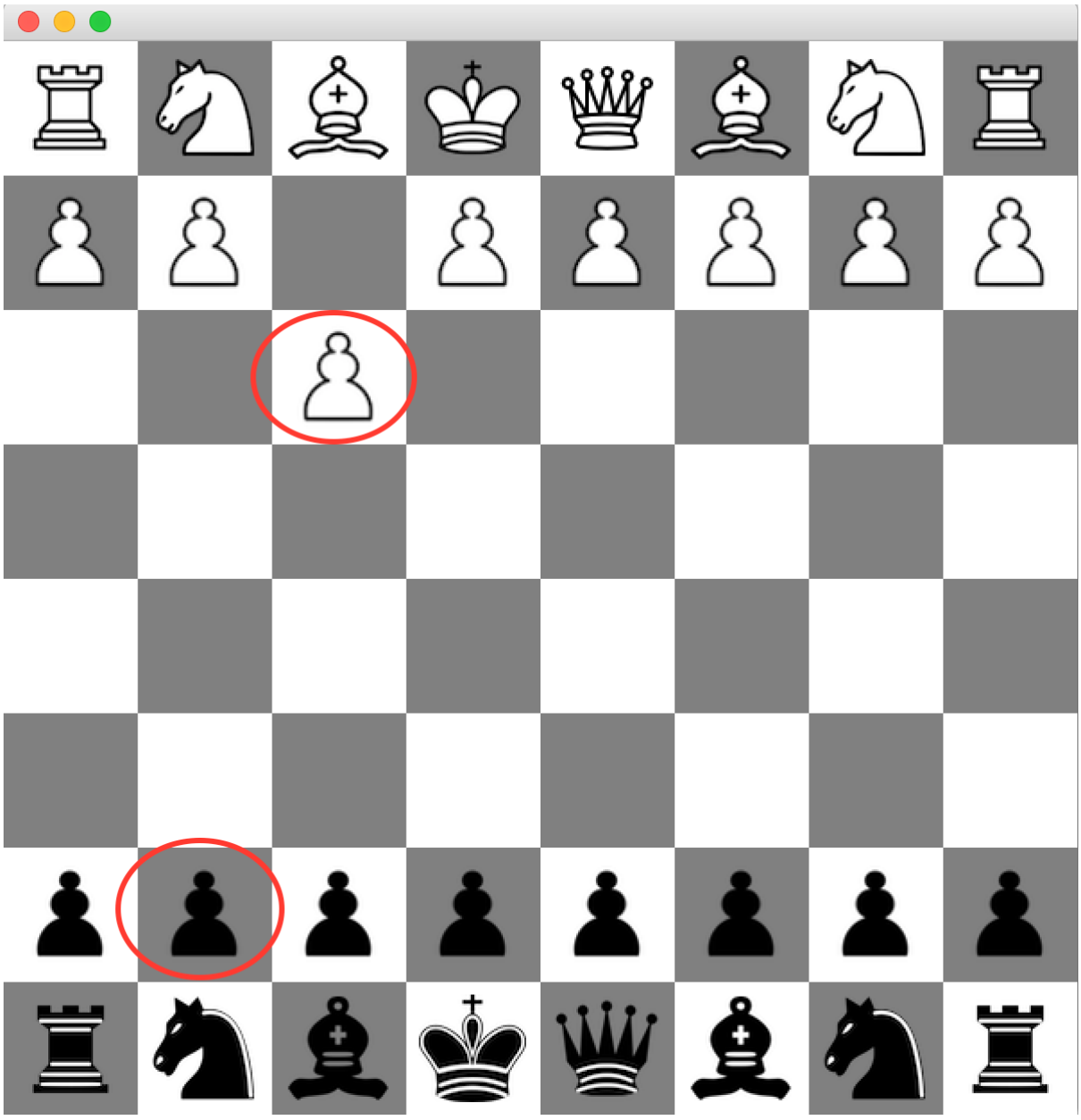

Figure 3: Second issue with the Pawn movements

Pawns can take pieces that are located in a diagonal position from its starting location, but only one square along a diagonal. From Figure 3 you need to restrict the distance that the pawn can move when taking an enemy piece.

Hint 3: The highlighted white pawn should be able to take the either of the highlighted black pieces.

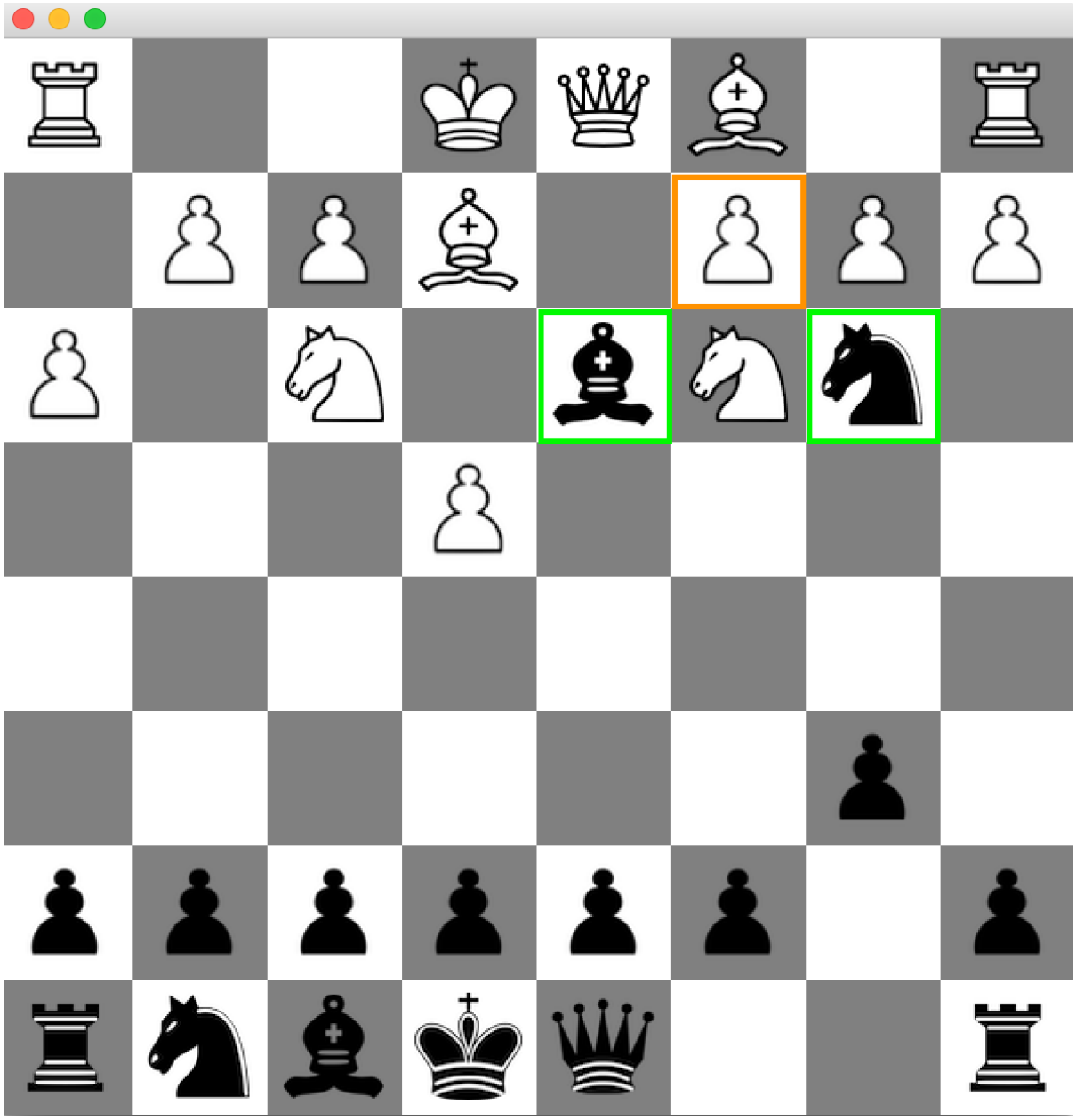

Figure 4: White pawns attacking black pieces

In Figure 4 above the highlighted pawn should be able to take either of the two highlighted black pieces. Don't worry, when you examine the code for when the pawn is making any move other than its first move the logic / code you need is there.

What else can you find??

Things to complete in the First Week:

1. Fix the identified issues above with the White pawn movements.

2. Enable the Black Pawn to move like the White Pawns.

But first let us consider the code that we have that makes the White Pawn move as seen below in Code
Snippet 1.

```
197     if(pieceName.equals("WhitePawn")){
198       if(startY == 1){
199         if((startX == (e.getX()/75))&&((((e.getY()/75)-startY)==1)||((e.getY()/75)-startY)==2)){
200           if((((e.getY()/75)-startY)==2)){
201             if((!piecePresent(e.getX(), (e.getY())))&&(!piecePresent(e.getX(), (e.getY()+75)))){
202               validMove = true;
203             }
204             else{
205               validMove = false;
206             }
207           }
208           else{
209             if((!piecePresent(e.getX(), (e.getY())))){
210               validMove = true;
211             }
212             else{
213               validMove = false;
214             }
215           }
216         }
217         else{
218           validMove = false;
219         }
220       }
221       else{  // The pawn is taking its second move...
```

Code Snippet 1: Understanding the Pawn Movements

Line 197: This condition uses the name of the piece to allow the game create the conditions for that particular piece to move. Here we are saying if the piece is a "White Pawn" do something…

Line 198: The pawn piece in Chess can move either one square or two squares on its first move. We need a way to determine if the pawn is making its first move. You can solve this condition many ways but here we are using the starting position of the piece to determine if it has previously moved. When a user clicks on a piece the variable *startY* and *startX* gives us a location for the piece.

Line 199: This looks a bit more complicated than it is. In English we would say "If the starting position of the x column is the same as the finishing column and the new y coordinate has moved either one or two squares, we **may** have a valid move". Here we are using the the mouseEvent *e* when the user tries to release a piece to get the location of were the piece has been dropped onto the board.

Line 200: Okay so we have moved inside the condition on line 199. The condition on line 200 asks if the piece has moved two squares, if not the code continues for a single square movement on line 208.

Line 201: Now we need to check if we have a piece on the square that we are moving to, we do this by using the method "piecePresent", which returns true if there is a piece present on a giving square or false if there is nothing on the square.

Make sure that both Pawns can make all possible pawn movements prior to moving onto Knight Moves.

Getting Started on making the Black Pawn move:

Let the game accept all Black Pawn movements as follows:

```
if(pieceName.equals("BlackPawn")){
  validMove = true;
}
else if(pieceName.equals("WhitePawn")){
  if(startY == 1){
```

Code Snippet 2: Accepting Black Pawn Movements

Essentially all we are doing here is saying if the piece name that is being moved is "BlackPawn" then set validMove equal to true (this is the Boolean variable that actually makes the move!). Once this logic is established, if you recompile your .java file and run the program, you should be able to move the Black Pawns as seen below in Figure 5.

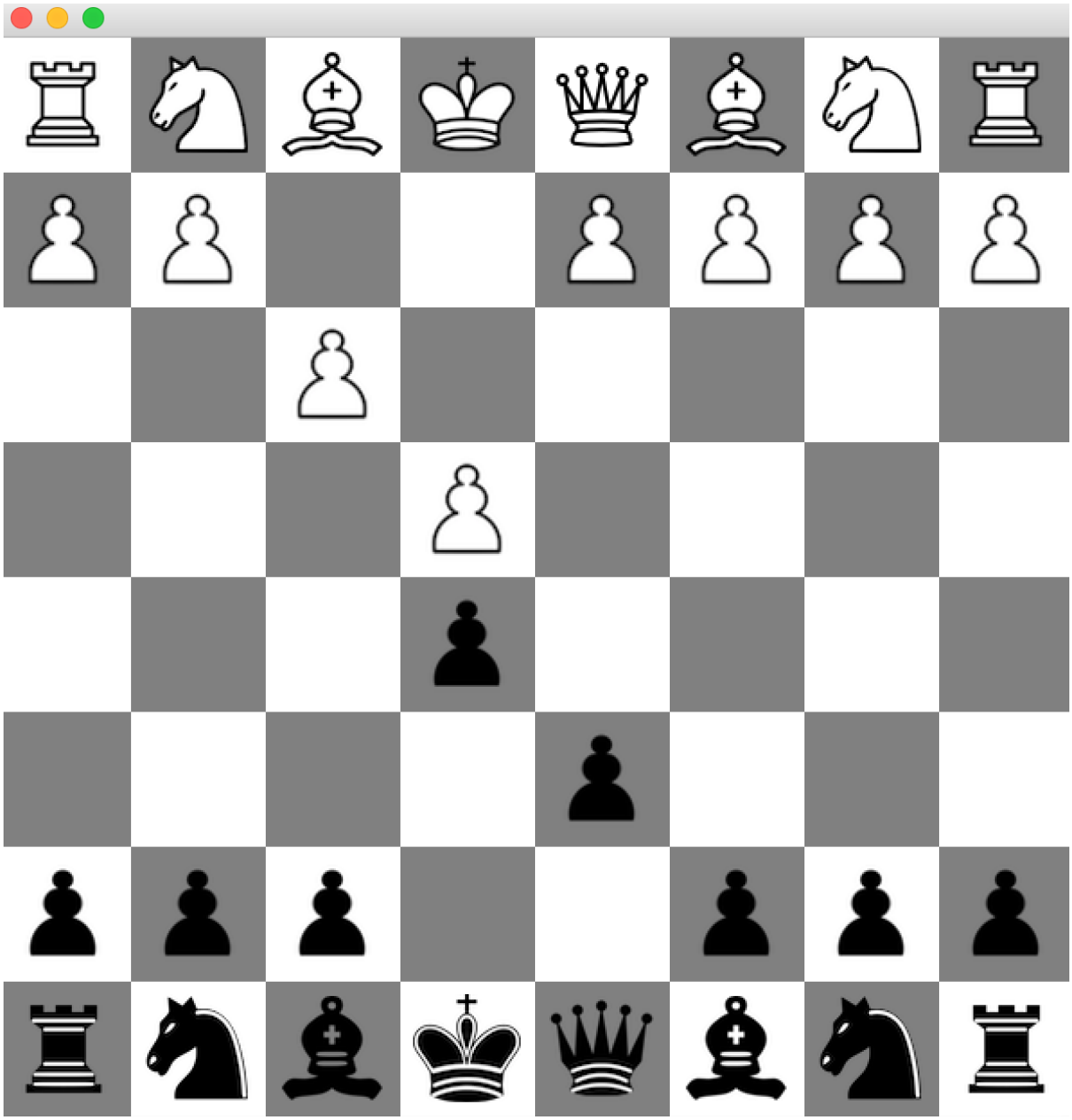

Figure 5: Black Pawns moving

Once the BlackPawn can move all that is required is to restrict the movements to make the piece operate like a pawn. It can be seen in Code Snippet 3 that the restrictions are being added to ensure that the piece acts like a pawn. Two variables have been added landingX and landingY which give the coordinates of were the piece is being dropped onto the board. In English we would say something like:

If the pawn is at the starting position

If the pawn is not being moved in the x direction and the pawn is either being moved one or two squares in the y direction

This is cool, move the piece!

Else

This isn't a proper move

Else if the pawn has already been moved, i.e. it is not at the starting position

If the pawn is not being moved in the x direction and if it is only being moved one square in the y direction

This is cool, move the piece!

Else

This isn't a valid move

Code Snippet 3 below translates this logic into Java, see below.

```
205      if(pieceName.equals("BlackPawn")){
206        /* The pawn can move either two or one squares */
207        if(startY == 6){
208          if((startX == landingX)&&(((startY-landingY)== 1)||(startY-landingY)== 2)){
209                validMove = true;
210          }
211          else{
212               validMove = false;
213          }
214        }
215        else{ // the piece is not in the starting position...
216          if((startX == landingX)&&(((startY-landingY)== 1))){
217                validMove = true;
218          }
219          else{
220                validMove = false;
221          }
222        }
223      }
224      else if(pieceName.equals("WhitePawn")){
```

Code Snippet 3: Restricting the movements of the Black Pawns

So we now have some basic constructs working for the Black Pawn. But if there is a piece in the way along the y direction we cannot make the move. Code Snippet 4 caters for this case, see below. This code should be inserted in each place in Code Snippet 3 were we have the expression *validMove = true;* This helps us incrementally build logic into the solution and you should recompile your code after each change and run the program to check if everything is working fine.

```
if(!piecePresent(e.getX(), e.getY())){
  validMove = true;
}
else{
  validMove = false;
}
```

Code Snippet 4: Checking if there is a piece present on a square

So how does this work. Well we see that we are calling a method called *piecePresent* and passing in some coordinates. The code is being executed in the *mouseReleased(MouseEvent e)* method and we can capture the location of the position using the *e.getX()* and *e.getY()* methods. Code Snippet 5 shows the *piecePresent* method below. It can be seen here that when this method is called, we get the component that resides at the coordinates that are passed into the method and simply check if the component is a *JPanel*. If it is, then there is nothing on the identified square. If its not an instance of a *JPanel* it would be a *JLabel*, which implies that there is a piece on the square.

```
private Boolean piecePresent(int x, int y){
  Component c = chessBoard.findComponentAt(x, y);
  if(c instanceof JPanel){
    return false;
  }
  else{
    return true;
  }
}
```

Code Snippet 5: Understanding the piecePresent Method

So now we can't move the Pawn to a square were there is a piece in the way and we can't jump over a piece. However, the Pawn can take an opponent's piece if the piece is in a diagonal position, see Figure 6 below for an example.

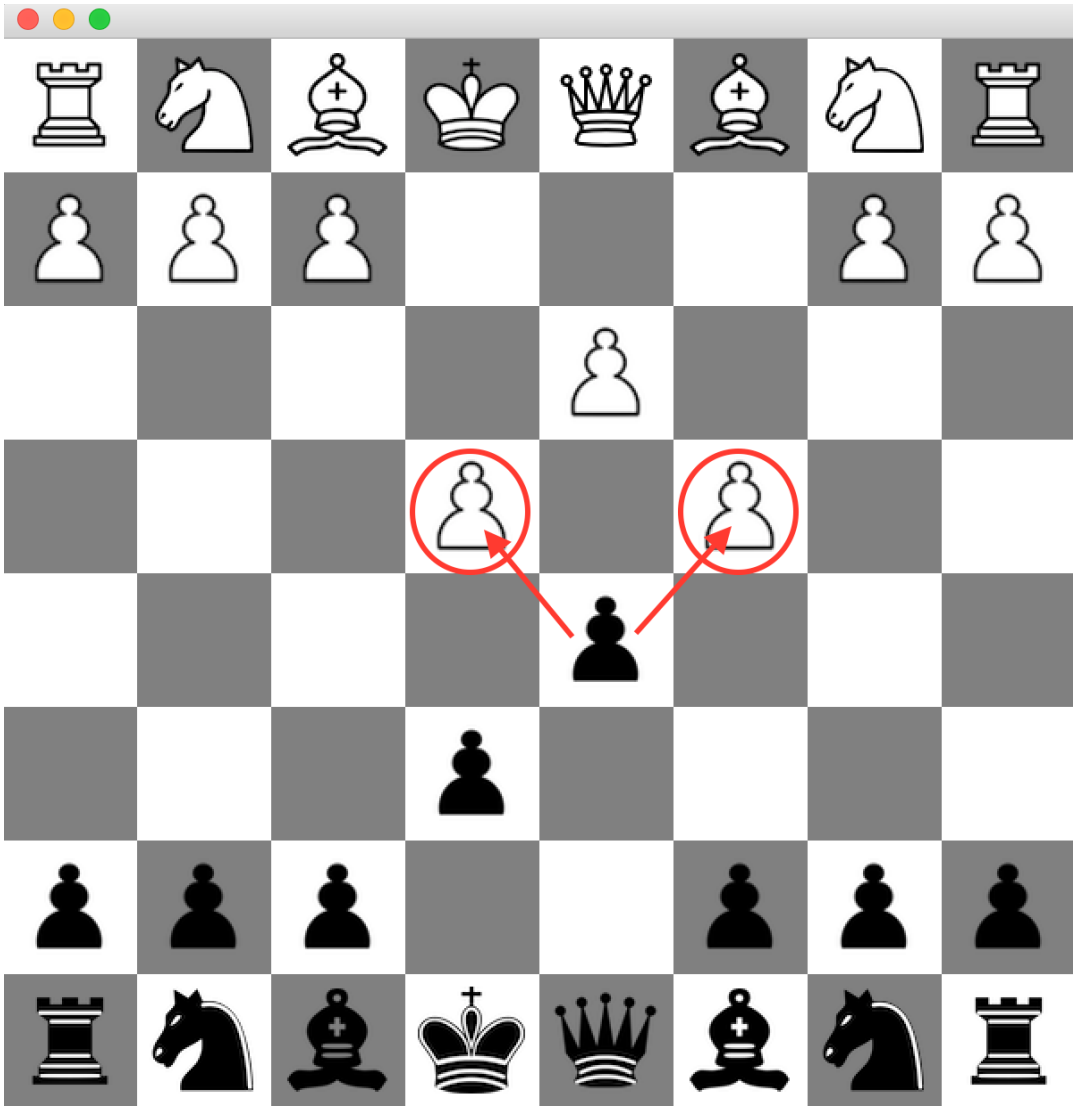

Figure 6: Pawn attacking movements

We need to create a method called *checkBlackOponent* that takes as an argument the coordinates of a position and checks if there is an opponent's piece present on the square. In the initial sample ChessProject file there is a sample *checkWhiteOponent* method, however this has been adapted for checking Black opponents in Code Snippet 6 below. This method simply checks to see if the *JLabel* on a given component contains the text *White* it returns true.

```
private Boolean checkBlackOponent(int newX, int newY){
  Boolean oponent;
  Component c1 = chessBoard.findComponentAt(newX, newY);
  JLabel awaitingPiece = (JLabel)c1;
  String tmp1 = awaitingPiece.getIcon().toString();
  if(((tmp1.contains("White")))){
    oponent = true;
  }
  else{
    oponent = false;
  }
  return oponent;
}
```

Code Snippet 6: Method to check if there is a White piece on a given square

So having the functionality to check if there is White piece on a giving square we need to modify our existing code for the Black Pawn as seen in Code Snippet 7 below. The first line of code checks if the square were the pawn is being put back onto the board is one diagonal to the left or the right of the starting square. If it is, firstly we need to check if there is a piece on the square that we want to move to, then we need to check if that piece is an opponent's piece.

```
else if((Math.abs(startX-landingX)==1)&&(((startY-landingY)== 1))){
     if(piecePresent(e.getX(),e.getY())){
         if(checkBlackOponent(e.getX(),e.getY())){
           validMove = true;
           if(landingY == 0){
             progression = true;
           }
         }
         else{
           validMove = false;
         }
     }
     else{
         validMove = false;
     }
  }
```

Code Snippet 7: Pawn takes a piece

As we can see above in Code Snippet 7, just after the validMove = true, we check if the landing position of the y coordinate is equal to 0. If this is we set a Boolean flag equal to true. This allows

us to change the Pawn into another piece as seen in Code Snippet 8 below (here we just change the piece into a Black Queen!).

```
else{
  if(progression){
    int location = 0 + (e.getX()/75);
    if (c instanceof JLabel){
      Container parent = c.getParent();
      parent.remove(0);
      pieces = new JLabel( new ImageIcon("BlackQueen.png") );
      parent = (JPanel)chessBoard.getComponent(location);
      parent.add(pieces);
    }
  }
  else if(success){
```

Code Snippet 8: Promoting a Pawn to a Black Queen

If you have implemented to here, that's great. You should at this stage:

- Fixed the White Pawn Movements
- Understand the following methods; *checkBlackOponent, checkWhiteOponent, piecePresent, e.getX(), e.getY()…etc.*
- Understand exactly how the *mouseReleased(MouseEvent e)* method is working to be able to modify it the way you need to get other pieces working / moving!
- Understand the logic of when a pawn reaches the opponents side of the board to be able to change into another piece.

Once you understand these components, getting the rest of the pieces moving isn't that hard!

```
    else if(pieceName.equals("BlackPawn")){
    /* The pawn can move either two or one squares */
    if((startY == 6)&&(startX == landingX)&&(((startY-landingY)== 1)||(startY-landingY)== 2)){
        /* If there is a piece in the way */
        if(!piecePresent(e.getX(), e.getY())){
          validMove = true;
        }
        else{
          validMove = false;
        }
    }
    else if((Math.abs(startX-landingX)==1)&&(((startY-landingY)== 1))){
          if(piecePresent(e.getX(),e.getY())){
              if(checkBlackOponent(e.getX(),e.getY())){
                validMove = true;
                if(landingY == 0){
                  progression = true;
                }
              }
              else{
                validMove = false;
              }
          }
          else{
              validMove = false;
          }
      }
      else if((startY != 6)&&((startX == landingX)&&(((startY-landingY)== 1)))){
        /* If there is a piece in the way */
        if(!piecePresent(e.getX(), e.getY())){
          validMove = true;
          if(landingY == 0){
            progression = true;
          }
        }
        else{
          validMove = false;
        }
      }
      else{
          validMove = false;
      }
  }
```

Code Snippet 9: Complete movements for Black Pawn

Knight Movements

Okay, now that you are considering the constructs for allow the knight to move, I know that you understand the codebase. First step lets just allow all Knight Moves (just like how we started with the Black Pawn). Just a little note, we don't need to create separate conditions for each colour for the Knight as they move in the same patterns, unlike the Black and White Pawns.

```
if(pieceName.contains("Knight")){
  validMove = true;
}
else if(pieceName.equals("BlackPawn")){
```

Code Snippet 10: Allowing all Knights to be able to move

Once we have enable the Knights, they can move to any square, including taking their own pieces, see below (Figure 7).

Figure 7: Knights attacking their own pieces

So like with the Black Pawn we need to understand were the piece is being returned to the board. You should already have the following two variables setup in the mouseReleased method.

```
int landingY = e.getY()/75;
int landingX = e.getX()/75;
```

We need to make sure that the piece is being put back on the board...if its not being on the board why would we want to check anything else!

```
if(((landingX < 0) || (landingX > 7))||((landingY < 0)||(landingY > 7))){
    validMove = false;
}
else{
```

We need to build a valid move for the Knight. The knight can move in an L shape. So we need to map out all possibilities for any L shape from any position. Consider Figure 7 below:

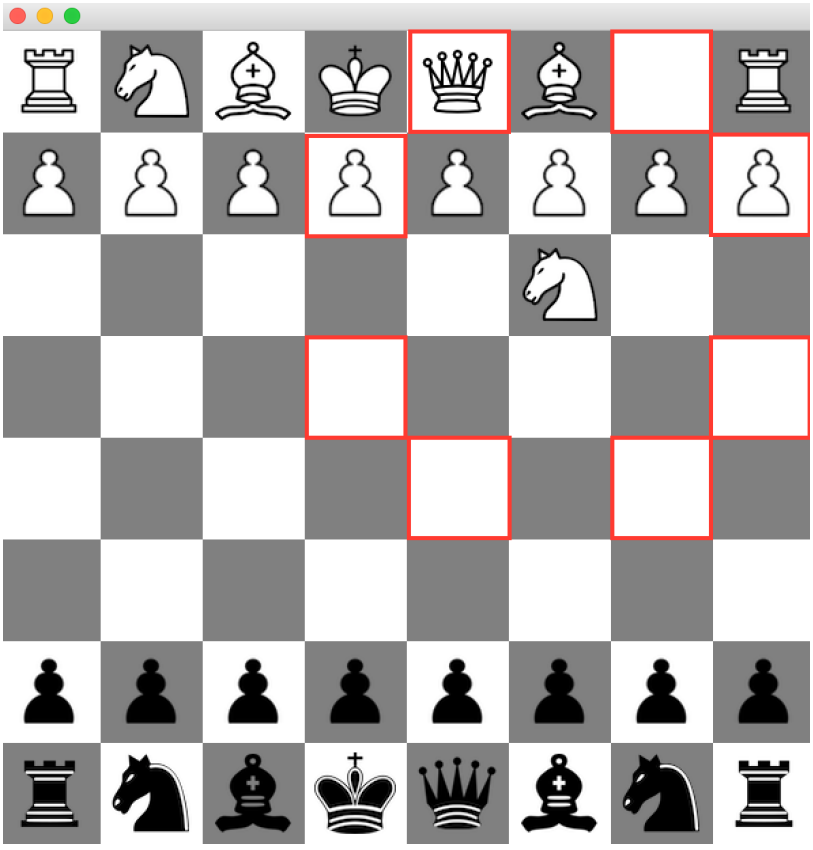

Figure 8: Understanding how the Knight moves

It can be seen in Figure 8 that the White Knight can potentially move into any of the red boxes highlighted, however it cannot take its own piece. But first we need to map out all the possible Knight moves, as seen in Code Snippet 11.

```
if(pieceName.contains("Knight")){
    if(((landingX < 0)||(landingX > 7))||((landingY < 0)||landingY > 7)){
      validMove = false;
    }
    else{
      if(((landingX == startX+1) && (landingY == startY+2))||((landingX == startX-1) && (landingY == startY+2))||((landingX ==
      startX+2) && (landingY == startY+1))||((landingX == startX-2) && (landingY == startY+1))||((landingX == startX+1) &&
      (landingY == startY-2))||((landingX == startX-1) && (landingY == startY-2))||((landingX == startX+2) && (landingY ==
      startY-1))||((landingX == startX-2) && (landingY == startY-1))){
        validMove = true;
      }
      else{
        validMove = false;
      }
    }
  }
```

Code Snippet 11: Implementing the knight moves

Now, our knight can move so we are all happy! But that is not good enough, as we need to be able to take an enemy piece when making the move. It is important to note that the Knight doesn't need to worry about pieces in the way while making the move as the piece jumps to the square it is moving to. We do need to consider that when the piece is returning to the board there could be a piece in the way: either our own piece in which case we cannot make the move or an enemy piece which we then need to take, see Code Snippet 12.

```
if(piecePresent(e.getX(),(e.getY()))){
  if(pieceName.contains("White")){
    if(checkWhiteOponent(e.getX(), e.getY())){
      validMove = true;
    }
    else{
      validMove = false;
    }
  }
  else{
    if(checkBlackOponent(e.getX(), e.getY())){
      validMove = true;
    }
    else{
      validMove = false;
    }
  }
}
else{
  validMove = true;
}
```

Code Snippet 12: Making sure that the Knight cannot take his own piece

Great, we now have the Knights moving the way they should (see complete code segment for the Knight in Code Snippet 13 below)! The next section shows how to get the Bishop moving around the board. The Bishop is a long range piece and is usually very effective as the game opens up.

```
else if(pieceName.contains("Knight")){
    if(((landingX < 0)||(landingX > 7))||((landingY < 0)||landingY > 7)){
      validMove = false;
    }
    else{
      if(((landingX == startX+1) && (landingY == startY+2))||((landingX == startX-1) && (landingY ==
      startY+2))||((landingX == startX+2) && (landingY == startY+1))||((landingX == startX-2) && (landingY ==
      startY+1))||((landingX == startX+1) && (landingY == startY-2))||((landingX == startX-1) && (landingY ==
      startY-2))||((landingX == startX+2) && (landingY == startY-1))||((landingX == startX-2) && (landingY ==
      startY-1))){
        if(piecePresent(e.getX(),(e.getY()))){
          if(pieceName.contains("White")){
            if(checkWhiteOponent(e.getX(), e.getY())){
              validMove = true;
            }
            else{
              validMove = false;
            }
          }
          else{
            if(checkBlackOponent(e.getX(), e.getY())){
              validMove = true;
            }
            else{
              validMove = false;
            }
          }
        }
        else{
          validMove = true;
        }
      }
      else{
        validMove = false;
      }
    }
  }
```

Code Snippet 13: Complete Code segment for the Knight Moves

Bishop Movements

Pythagoras can't help here. If we consider that the bishop moves along a diagonal (see Figure 8 below) and each square piece is the same size...this means that the difference between the (x, y) and (x1, y1) coordinates have to be equal for each point. You just need to ignore the sign of the difference between the squares.

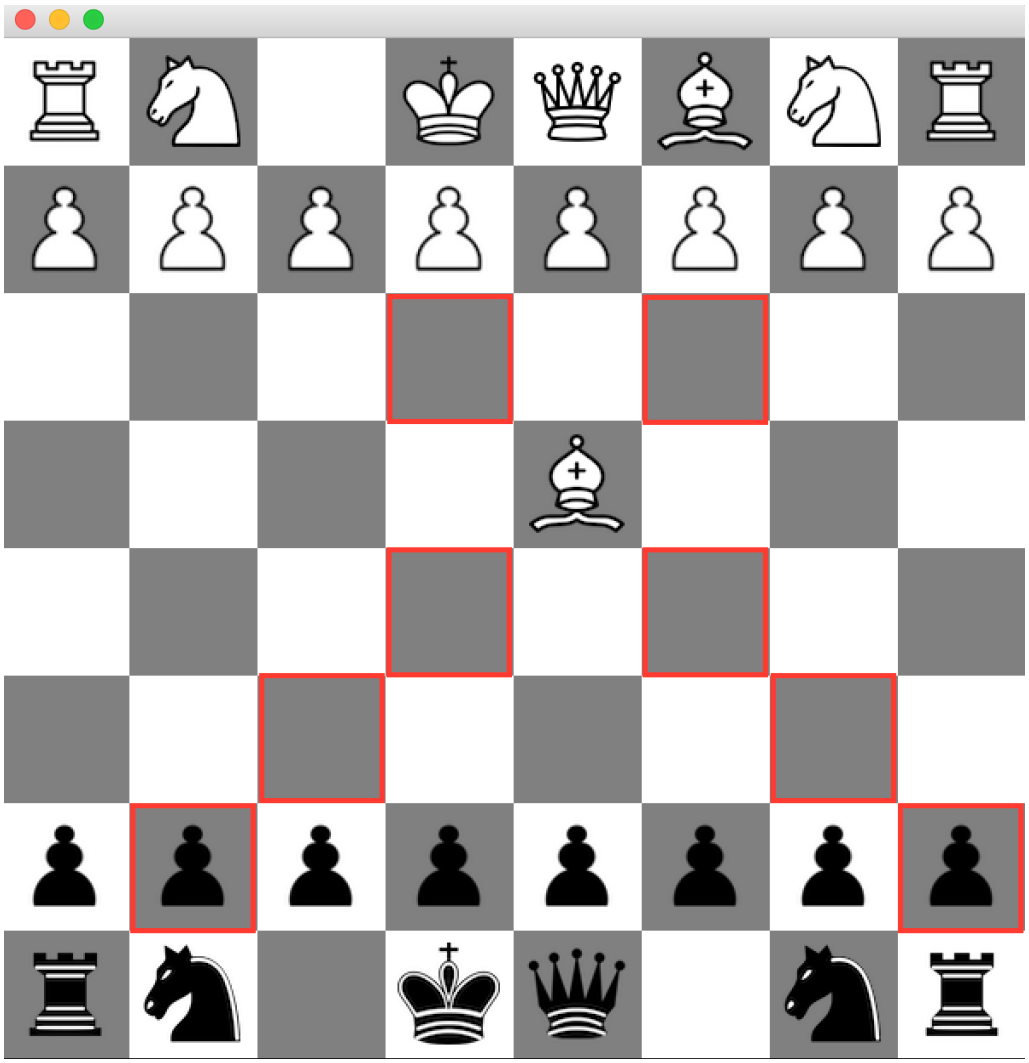

Figure 9: Bishop Movements - equal x and y distances, moves the Bishop along the diagonal

We also have to check every square along the diagonal to make sure that there is no piece in the way....this sounds like a loop with a piecePresent method being called on each iteration.

```
if(Math.abs(startX-landingX)==Math.abs(startY-landingY)){
  if((startX-landingX < 0)&&(startY-landingY < 0)){
    for(int i=0; i < distance;i++){
      if(piecePresent((initialX+(i*75)), (initialY+(i*75)))){
        inTheWay = true;
      }
    }
  }
  else if((startX-landingX < 0)&&(startY-landingY > 0)){
    for(int i=0; i < distance;i++){
      if(piecePresent((initialX+(i*75)), (initialY-(i*75)))){
        inTheWay = true;
      }
    }
  }
  else if((startX-landingX > 0)&&(startY-landingY > 0)){
    for(int i=0; i < distance;i++){
      if(piecePresent((initialX-(i*75)), (initialY-(i*75)))){
        inTheWay = true;
      }
    }
  }
  else if((startX-landingX > 0)&&(startY-landingY < 0)){
    for(int i=0; i < distance;i++){
      if(piecePresent((initialX-(i*75)), (initialY+(i*75)))){
        inTheWay = true;
      }
    }
  }
```

Code Snippet 14: Checking along the diagonal for a Bishop to see if there is a piece in the way

It can be seen in Code Snippet 14 that we essentially have four conditions to determine the direction of the diagonal that the Bishop is intending to move along. We need to determine this direction as we need to check each square along the diagonal to make sure that there is no piece in the way. We are using here the values for the *initialX* and *initialY*, these values are calculated when the piece is selected in the *mousePressed(MouseEvent e)* method. If we find a piece along the diagonal we set a Boolean variable to be true.

This is cool now, all we have to do is check if the Boolean variable was set to true, if so there is a piece in the way and the move is considered an invalid move. If the Boolean is false, this means that we still have a potential good move and we need to check the landing square of where the piece is being moved to. If this is an enemy piece we should be able to take it, however it is our own piece we should be stopped from taking it, see Code Snippet 15 below:

```
if(inTheWay){
  validMove = false;
}
else{
  if(piecePresent(e.getX(), (e.getY()))){
    if(pieceName.contains("White")){
      if(checkWhiteOponent(e.getX(), e.getY())){
        validMove = true;
      }
      else{
        validMove = false;
      }
    }
    else{
      if(checkBlackOponent(e.getX(), e.getY())){
        validMove = true;
      }
      else{
        validMove = false;
      }
    }
  }
  else{
    validMove = true;
  }
}
```

Code Snippet 15: Checking if the Bishop can take an opponent piece

Okay, so great progress is being made. To recap, to get the Bishop moving we did the following:

- Checked if the *pieceName* contained the string Bishop
- Created a Boolean variable called *inTheWay*
- Checked if the piece was being put back on the Chessboard
- Determined if the Bishop was moving along a diagonal
- Determined which direction the diagonal was moving and checked each square along this diagonal to determine if there is a piece in the way
- Made sure that the Bishop could take only an opponent piece

The complete Code Snippet for the Bishop movements is seen in Code Snippet 16 below.

```
if(pieceName.contains("Bishop")){
    Boolean inTheWay = false;
    int distance = Math.abs(startX-landingX);
    if(((landingX < 0) || (landingX > 7))||((landingY < 0)||(landingY > 7))){
      validMove = false;
    }
    else{
      validMove = true;
      if(Math.abs(startX-landingX)==Math.abs(startY-landingY)){
        if((startX-landingX < 0)&&(startY-landingY < 0)){
          for(int i=0; i < distance;i++){
            if(piecePresent((initialX+(i*75)), (initialY+(i*75)))){
              inTheWay = true;
            }
          }
        }
        else if((startX-landingX < 0)&&(startY-landingY > 0)){
          for(int i=0; i < distance;i++){
            if(piecePresent((initialX+(i*75)), (initialY-(i*75)))){
              inTheWay = true;
            }
          }
        }
        else if((startX-landingX > 0)&&(startY-landingY > 0)){
          for(int i=0; i < distance;i++){
            if(piecePresent((initialX-(i*75)), (initialY-(i*75)))){
              inTheWay = true;
            }
          }
        }
        else if((startX-landingX > 0)&&(startY-landingY < 0)){
          for(int i=0; i < distance;i++){
            if(piecePresent((initialX-(i*75)), (initialY+(i*75)))){
              inTheWay = true;
            }
          }
        }

        if(inTheWay){
          validMove = false;
        }
        else{
          if(piecePresent(e.getX(), (e.getY()))){
            if(pieceName.contains("White")){
              if(checkWhiteOponent(e.getX(), e.getY())){
                validMove = true;
              }
              else{
                validMove = false;
              }
            }
            else{
              if(checkBlackOponent(e.getX(), e.getY())){
                validMove = true;
              }
              else{
                validMove = false;
              }
            }
          }
          else{
            validMove = true;
          }
        }
      }
      else{
        validMove = false;
      }
    }
  }
```

Code Snippet 16: Complete Bishop Code

Rook Movements

The Rook is a piece that moves in either a horizontal or vertical movement. It can move any number of squares but cannot pass through a piece. Like all Chess pieces it can take an opponent piece but not its own piece.

Firstly, we need to make sure that the piece is being put back onto the Chess board. Essentially what we are saying here is if the piece is not being put back onto the board its not a valid move as we have added for all other pieces thus far. The first condition that we need to recognise that a move might be a possible Rook movement is that we will have a change in the X or Y coordinates but not both, see Code Snippet 17 below.

```
if(((Math.abs(startX-landingX)!=0)&&(Math.abs(startY-landingY) == 0))||
((Math.abs(startX-landingX)==0)&&(Math.abs(landingY-startY)!=0)))
{

  // Possible Move
}
```

Code Snippet 17: Checking to see if the Rook has either moved along the X or Y axis but not both

Okay, so things are simple now. The Rook is going to move. Well if its an X Movement it moves left or right. If it's a Y movement, it moves up or down. See Figure 9 below for Rook movements.

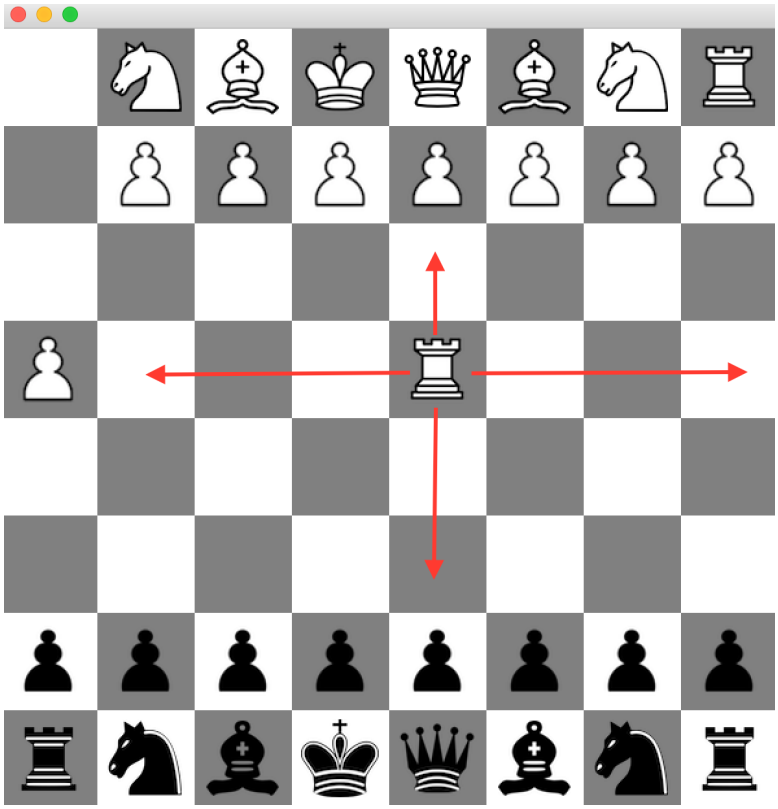

Figure 10: Possible Rook movements

The first condition that we have below in Code Snippet 18 caters for when we have a horizontal movement. This movement either goes from left to right or right to left. We need to determine the direction of the movement as we have to make sure that there are no pieces in between the starting position and the landing position of the piece being moved.

```
if(Math.abs(startX-landingX)!=0){
  int xMovement = Math.abs(startX-landingX);
  if(startX-landingX > 0){
    for(int i=0;i < xMovement;i++){
      if(piecePresent(initialX-(i*75), e.getY())){
        intheway = true;
        break;
      }
      else{
        intheway = false;
      }
    }
  }
  else{
    for(int i=0;i < xMovement;i++){
      if(piecePresent(initialX+(i*75), e.getY())){
        intheway = true;
        break;
      }
      else{
        intheway = false;
      }
    }
  }
}
```

Code Snippet 18: Rook Movement along the X axis

Obviously, the else condition for the above if caters for the horizontal movements. Once you integrate the code for the else part we now have the constructs to make the Rook move. So let us recap of how we got the Rook moving:

- Checked if the *pieceName* contained the string Rook
- Created a Boolean variable called *intheWay*
- Checked if the piece was being put back on the Chessboard
- Determined if the Rook was moving along a vertical or horizontal path
- Determined which direction the Rook was moving and checked each square along this path to determine if there is a piece in the way
- You now need to make sure that the Rook can take an opponent piece and set the movement to be a validMove (i.e. *validMove = true;*)

If you are having difficulties completing the code to support the movements of the Rook, see Code Snippet 19 below.

```
if(pieceName.contains("Rook")){
    Boolean intheway = false;
    if(((landingX < 0) || (landingX > 7))||((landingY < 0)||(landingY > 7))){
      validMove = false;
    }
    else{
      if(((Math.abs(startX-landingX)!=0)&&(Math.abs(startY-landingY) == 0))|| ((Math.abs(startX-landingX)==0)&&(Math.abs(landingY-startY)!=0)))
      {
        if(Math.abs(startX-landingX)!=0){
          int xMovement = Math.abs(startX-landingX);
          if(startX-landingX > 0){
            for(int i=0;i < xMovement;i++){
              if(piecePresent(initialX-(i*75), e.getY())){
                intheway = true;
                break;
              }
              else{
                intheway = false;
              }
            }
          }
          else{
            for(int i=0;i < xMovement;i++){
              if(piecePresent(initialX+(i*75), e.getY())){
                intheway = true;
                break;
              }
              else{
                intheway = false;
              }
            }
          }
        }
        else{
          int yMovement = Math.abs(startY-landingY);
          if(startY-landingY > 0){
            for(int i=0;i < yMovement;i++){
              if(piecePresent(e.getX(),initialY-(i*75))){
                intheway = true;
                break;
              }
              else{
                intheway = false;
              }
            }
          }
          else{
            for(int i=0;i < yMovement;i++){
              if(piecePresent(e.getX(),initialY+(i*75))){
                intheway = true;
                break;
              }
              else{
                intheway = false;
              }
            }
          }
        }

        if(intheway){
          validMove = false;
        }
        else{
          if(piecePresent(e.getX(), (e.getY()))){
            if(pieceName.contains("White")){
              if(checkWhiteOponent(e.getX(), e.getY())){
                validMove = true;
              }
              else{
                validMove = false;
              }
            }
            else{
              if(checkBlackOponent(e.getX(), e.getY())){
                validMove = true;
              }
              else{
                validMove = false;
              }
            }
          }
          else{
            validMove = true;
          }
        }
    }
    else{
      validMove = false;
    }
  }
}
```

Code Snippet 19: Complete code for Rook movements

Queen Movements

The Queen is the most powerful piece on the board. The Queen can move in a horizontal, vertical or diagonal direction as long as there are no pieces in the way (see Figure 11 below). We already have these movements from coding the Bishop and the Rook. Code the Queen!

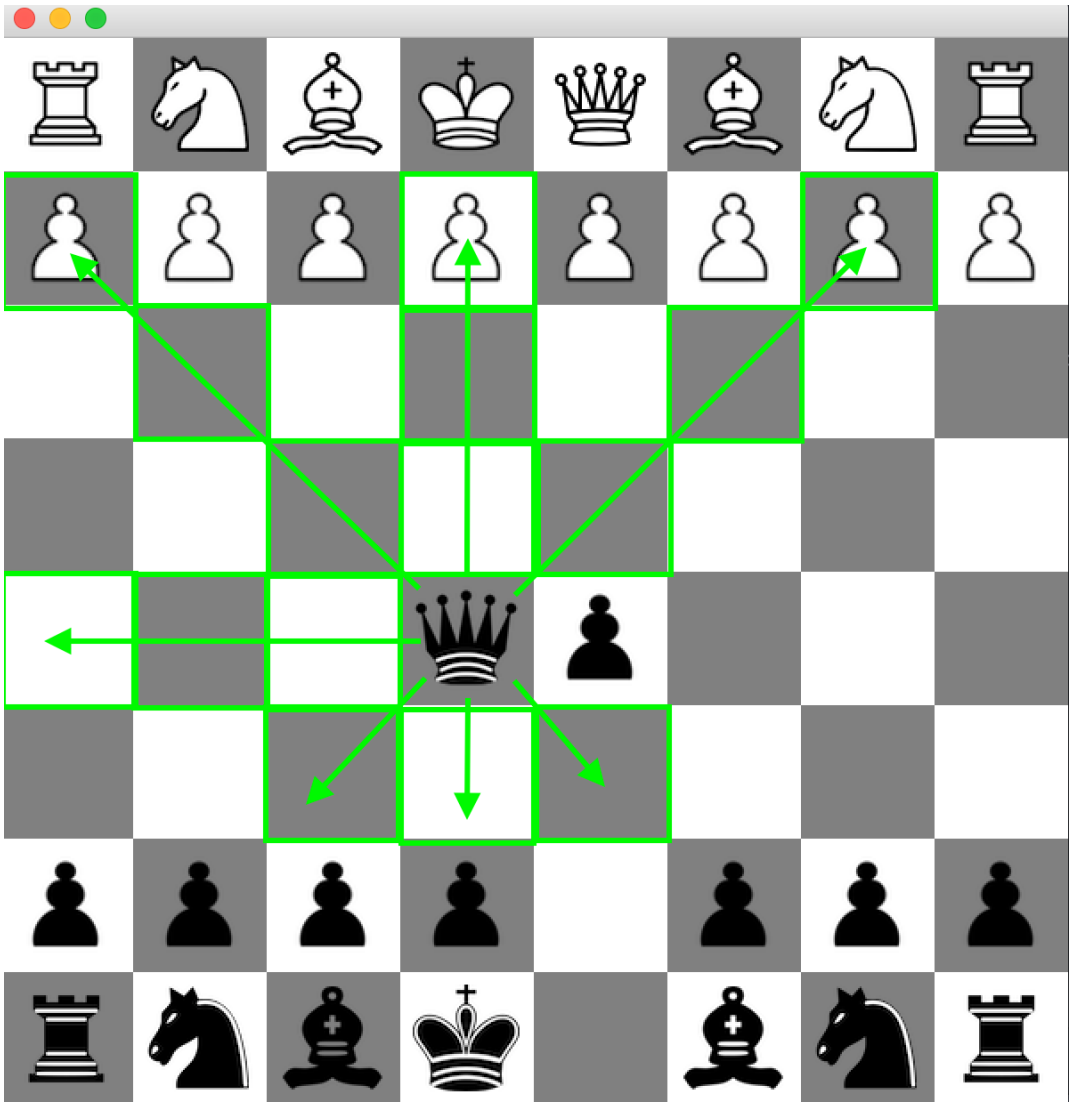

Figure 11: Queen movements

King Movements

The King is probably the hardest piece to move, so instead of trying to eat an elephant whole let us break up the task:

- First just make the King move by setting the ValidMove equal to true regardless of the where the king is being move to.
- Restrict the King to only move one square (see Figure 12 below)

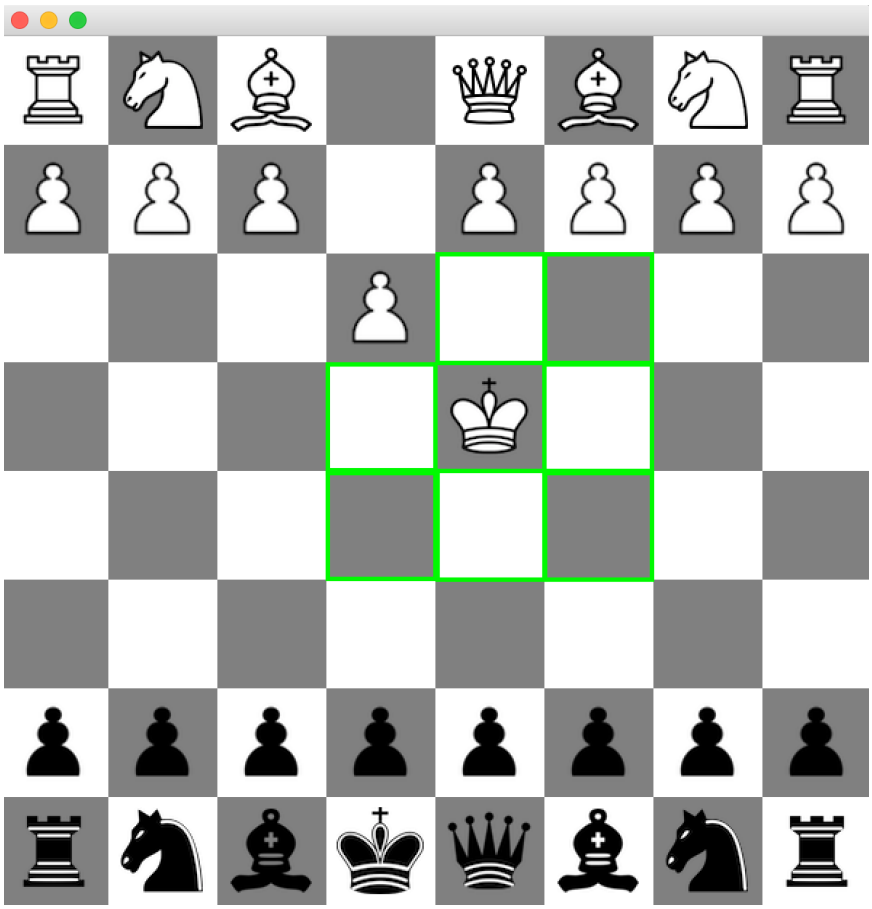

Figure 12: The King can only move one square at a time

- Make sure that the King cannot take its own piece but can take an opponent piece.

- The two Kings require at least one square between them...I would refactor this condition to be

the first control element, as you shouldn't be checking if you can take a piece and then figure out that the piece you are taking is adjacent to your enemy King (see Figure 13 for valid King movements).

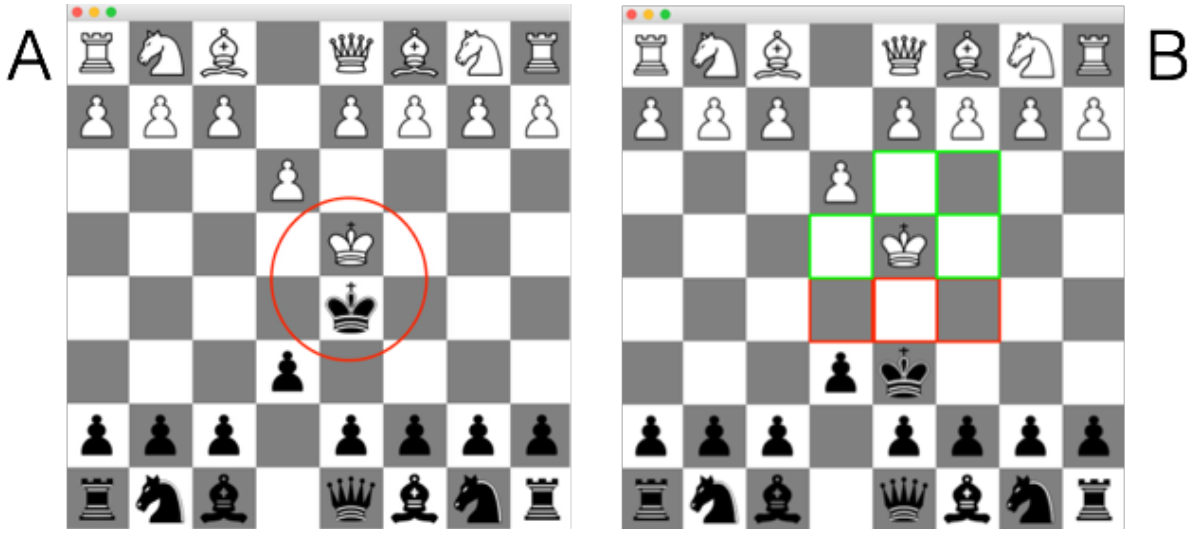

Figure 13: Part A shows two kings in adjacent squares, which should be avoided and part B shows all possible King movements for the White King

In addition to the movements above, the final condition that you should make sure is integrated is that you cannot move your king into a position that a piece is going to be directly attacking your King, i.e. you would be in Check! This will require you to create a method that loops through all the enemy chess pieces and finds out if they can see / attack the square you are moving too. This condition should be left to the second part of the project.

The game Chess finishes when a person is placed into Checkmate, which means your enemy's King is being attacked and cannot escape! For the first part of the project the game is won when somebody takes their opponent's King. You should inform the players who won, see Figure 14 below.

Figure 14: Dialogue to be shown when the game is won