A REPORT ON

CHESS GAME

(Java based GUI)

Submitted by:

Pawan Kumar

1120555

IT-4

2012-16

**DECLARATION**

I, hereby declare that the report of the project entitled **“CHESS GAME(Java based GUI)”** has not been presented as a part of any other academic work to get our degree or certificate except **National Institute of Technology, Kurukshetra** for the fulfillment of the requirements for the degree of Bachelor of Technology with Information Technology as major.

**ACKNOWLEDGEMENT**

The present report would not have been possible without the help; I have received from various quarters. I shall be failing in my duty if don’t acknowledge the help and guidance from these sources.

I extend my special thanks to employees of DLW EDP dept. and their supervisor, Mr. Deepak, for their benevolent guidance and kind cooperation throughout my training and for completing this project report. I also convey my special thanks to all other staff members. This guidance has helped me a lot in familiarizing with the company.

Last but not least, I am thankful to all for their sincere effort during training.

Pawan Kumar

Roll No. 112055

B.Tech 3rd Year

National Institute of Technology Kurukshetra

**INDEX**

Abstract

Introduction

Previous work

Initial phase and design

Actual design

Initiation specification

Challenges and reasons for modification

Action Listener

Usage

Model of listener

Challenges and modifications

Chess pieces

Pawn

Queen

Knight

King

Rook

Bishop

Check programming

Introduction

Implementation

Obstructions and modifications

Miscellaneous functions

Painter()

Sysexit()

Memory retention

Description

Implementation

Obstructions and modifications

Playing chess

Summary

Reference

**ABSTRACT**

This project implements a classic version of Chess with a Graphical User Interface. The Chess game follows the basic rules of chess, and all the chess pieces only move according to valid moves for that piece. This implementation of Chess is for two players (no Artiﬁcial Intelligence). It is played on an 8x8 checkered board, with a dark square in each player’s lower left corner. This project has been developed using java swing applet GUI. Although I was not able to design working artiﬁcial intelligence for the chess game in the time allowed, I researched few past implementations. I successfully created a GUI using JApplets and its container properties, as speciﬁed. Despite several unusual bugs in the GUI, this Chess program is a great, user-friendly game for two players.

**INTRODUCTION**

This project implements a classic version of Chess using Java Swing and a Graphical User Interface. The Chess game follows the basic rules of chess, and all the chess pieces only move according to valid moves for that piece. This implementation of Chess is for two players (no Artificial Intelligence). It is played on an 8x8 checkered board, with a dark square in each player’s lower left corner

**Previous Work**

Much previous research has been conducted in Chess artiﬁcial intelligence and creating a more realistic/intelligent Chess match. Although I am not able to design working artiﬁcial intelligence for the chess game in the time allowed, we researched the past implementations. I have used simple techniques here, action listener property of java is extensively used. Every piece of chess board is represented as image icons and methods used inside are abstracted and explained in further chapters. However, this technique is in its early stages, and requires that multiple assumptions and a complicated detection process.

**In further sections, I am going to discuss about design and working of this chess.**

**INITIAL PHASE AND DESIGN**

**Layout and initial design**

**Actual design**

As layout of this project, I chose grid layout and on every grid I embedded a content pane. Further, on each content pane, there is a button and on each button border layout is used so that it can take over whole space inside that panel. One important property that is used frequently in this program is getname() and setName() functions. In java, each element inside a program can be assigned a name. Name is of string data type. It can be extracted and change at any point of time. Other than that, other important feature used is foreground and background color property on button. Background color sets the background and foreground color sets the color of text that would be written over that button. In real chess board, squares are of black and white color alternatively, but for making board more attractive, I have used light gray instead of black.

**Initiation Specification**

When game initializes, First of all, applet main function app() is called. Following functions are performed in this app function: -

* Initialization of a root pane element which is later set as the root panel for project.
* A secondary JPanel initialization which is used as primary pane for all functions on board.
* JPanel mentioned in previous point is set to a 8x8 grid layout.
* A nested loop set which is used to embed a two dimensional array of JPanels on each section of grid.
* Assignment of name to all JPanels. They are actually serial number of panel. Starting from 0 to 63.
* Each frame is then picked up individually and assigned a button.
* All buttons are then painted using set background method in color white and black in alternative fashion to give it look of a chess board.
* Each button is then assigned a name and a foreground color. I used the foreground color to represent the color of chess piece which is there on that section.
* Pieces are then assigned on their rightful places by using setIcon() method to assign images and setName() function to assign name of that piece to that button for internal use . From 48 to 55, white pawns are assigned. From 8 to 15, black pawns are assigned. On 0,7,56 and 63, rooks are assigned and so on.
* For all black pieces, foreground color is black and for all white pieces, foreground color is white.
* All buttons are then made responsive by adding Action Listener to every one of them individually.
* Initialized root pane is added to main content pane and secondary JPanel is added on primary content pane.
* Used a JOptionPane message box, to show a welcome initial message.

Initial design of chess board and pieces is done in this way.

**Challenges and reason of modification**

Initial layout and designing of this game was a bit of a challenge. As we know, game of chess is a 8x8 grid and I had to make grid responsive. I knew I had to use grid layout but initial challenge with grid layout is that after you use first nested loop set, you cannot access elements again. So I used a two dimensional array of panels and assigned it to each section of grid so that it could be easily used. Other challenge was constant refreshing on page. Usage of root pane is a very basic technique used for quick refreshing of design frames. I used a root pane, assigned secondary frame on that root pane using setContentPane() method and based graphics of whole game on that secondary JPanel. Setting name on button did increase the no. of statements in program but it turned out that it gave greater flexibility in later stages of program. Internal usage of name and foreground color was extremely helpful.

**ACTION LISTENER**

**Usage**

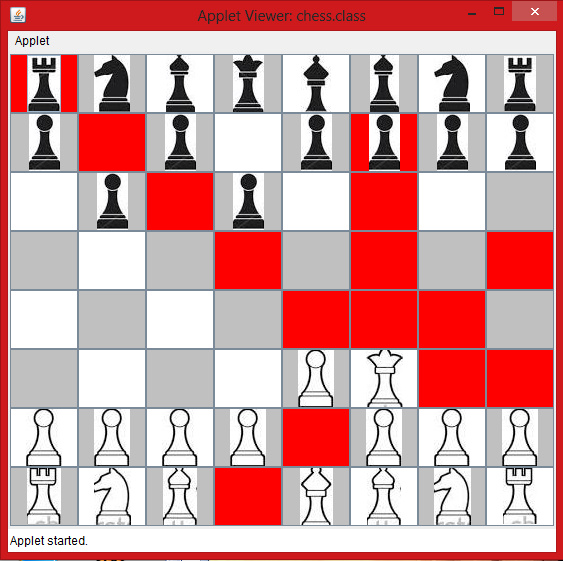
Action listeners are important utility in java. You can bind function with action listeners in such a way that whenever some event takes place for e.g. a button is clicked or a key is pressed, the bounded function is automatically called. As any developer could foresee, this is probably most important aspect of this project. All 64 buttons are assigned an action listener. In following project, almost everything is centered on listener method body. Work of action listener method is explained below.

**Model of action listener**

Following tasks are performed inside the action listener: -

* As soon as listener for some button is activated following things are calculated: -
  + 1. Source button
    2. Source panel
    3. Row and position of piece is deduced by simple arithmetic operations.
    4. Whether piece is white or black
* First thing that is figured out is whether there is a piece on that place or not.
* If some piece is there, then type of piece and color of piece is extracted using name and foreground color (see initial layout), next turn is also calculated. Details are stored in temporary variables. Status of applet window is updated in accordance to turn.
* Action listener method is programmed in such a way that every time action listener is called, turn of white and black pieces is alternatively set using a simple if-else mutual exclusion statement.
* Focus on window is requested so that window become active for user induced click events.
* A key listener is also added for memory element. This feature is further explained in memory retention chapter.
* Next step is extraction of type of piece and subsequent calling of appropriate function. Each type of piece and its programming is explained in further chapters.
* In mutually excluded parts, following things are programmed: -
  + - If the button which had been clicked has some piece over it and counter is set to 0, than it implies that user is in first phase of his move. Selected buttons identity, the piece over it (If any) and color of that piece is extracted and saved in a set of variables holding previous identity which is to be used in later phase. Scope of that piece is colored in red (see further scope of pieces chapter), counter is set to 1 and listener becomes inactive.
    - Next time when listener becomes active and observes that counter is set to 1. Then it prepares itself for second phase of move and stores all relevant information in a set of variables holding current identity. there can be two possible conditions now: -
      * User can click on an empty space or on chess piece of opponent, if that happens piece is moved to that button simply and all entries are overridden between previous and current position identity variable. Icon is also overridden between these two places.
      * If user clicks on a position which is out of scope i.e. which is not there in red background boundaries, whole grid is repainted again and user has to start from first phase again.
    - Turn is again updated after two phases of a move is over.

Image below shows the scope of a piece on chess board.



As shown in the image, user clicks on queen in first phase and immediately all possible moving places are painted in red. White pieces are not in scope as it is against the rules. In second phase, if user clicks inside the scope then chance will end otherwise user will have to start all over again from first phase.

**Challenges and modifications**

I handled following challenges in implementation of listeners:-

* Applet was very passive in nature. The problem with idea of painting the scope background button as red was that if user clicked on some piece outside the scope, everything needed to start right from the beginning of move i.e. phase 1. I thought only use of repaint() function is enough to set everything right, but it didn’t work. On little bit of research, I found out that only using repaint function was not enough, I had to use revalidate() function to make the grid active again.
* Clicking on any key didn’t work at first. It was quite a headache and problem lasted for a long, but later I found out that when applets are initialized, the focus of application is on initial JPanel and as soon as user presses some key, key is not applied to applet but to native window desktop. I discovered later that to get focus on java main applet window, there is a predefined function in java i.e. requestFocusInWindow ().

**CHESS PIECES**

**Pawn**

**E:\workspaces\chess\pawnw.jpg E:\workspaces\chess\pawnb.jpg**

**Introduction**

The **pawn** is the most frequent piece in the game of chess, and in most circumstances, also the weakest. It historically represents infantry, or more particularly, armed peasants or pike men. Each player begins a game of chess with eight pawns, one on each square of the rank immediately in front of the other pieces. (In algebraic notation, the white pawns start on a2, b2, c2, ..., h2, while black pawns start on a7, b7, c7, ..., h7.)

Individual pawns are referred to by the file on which they stand. For example, one speaks of "White's f-pawn" or "Black's b-pawn", or less commonly, “White's king bishop pawn" or "Black's queen knight pawn". It is also common to refer to a rook pawn, meaning any pawn on the a- or h-file, a knight pawn (on the b- or g-file), a bishop pawn (on the c- or f-file), a queen pawn (on the d-file), a king pawn (on the e-file), and a central pawn (on either the d- or e-file). I have further specified few more properties of a pawn in a chess game but I have only implemented single move forward and cross occupation move in my project.

**Scope**

Unlike the other pieces, pawns may not move backwards. Normally a pawn moves by advancing a single square, but the first time a pawn is moved, it has the option of advancing two squares. Pawns may not use the initial two-square advance to jump over an occupied square, or to capture. Any piece directly in front of a pawn, friend or foe, blocks its advance. In the diagram at right, the pawn on c4 may move to c5, while the pawn on e2 may move to either e3 or e4.

Unlike other pieces, the pawn does not capture in the same direction as it otherwise moves. A pawn captures diagonally, one square forward and to the left or right. In the diagram to the left, the white pawn may capture either the black rook or the black knight.

Another unusual move is the *en passant* capture. This arises when a pawn uses its initial move option to advance two squares instead of one, and in so doing passes over a square that is attacked by an enemy pawn. That enemy pawn, which would have been able to capture the moving pawn had it advanced only one square, is entitled to capture the moving pawn "in passing" *as if* it had advanced only one square. The capturing pawn moves into the empty square over which the moving pawn passed, and the moving pawn is removed from the board. In the diagram at right, the black pawn has just moved c7 to c5, so the white pawn may capture it by going from d5 to c6. The option to capture *en passant* must be exercised on the move immediately following the double-square pawn advance, or it is lost for the rest of the game. The *en passant* move was added to the pawn's repertoire in the 15th century to compensate for the then newly added two-square initial move rule. Without *en passant*, a pawn could simply march past squares guarded by opposing pawns; *en passant* preserves the restrictive ability of pawns that have reached the fifth rank.

**Promotion (special property)**

A pawn that advances all the way to the opposite side of the board (the opposing player's first rank) is *promoted* to another piece of that player's choice: a queen, rook, bishop, or knight of the same color. The pawn is immediately (before the opposing player's next move) replaced by the new piece. Since it is uncommon for a piece other than a queen to be chosen, promotion is often called "queening". When some other piece is chosen it is known as "underpromotion", and the piece selected is most often a knight, used to execute a checkmate or a fork giving the player a net increase in material compared to promoting to a queen. Underpromotion is also used in situations where promoting to a queen would give immediate stalemate.

**Implementation**

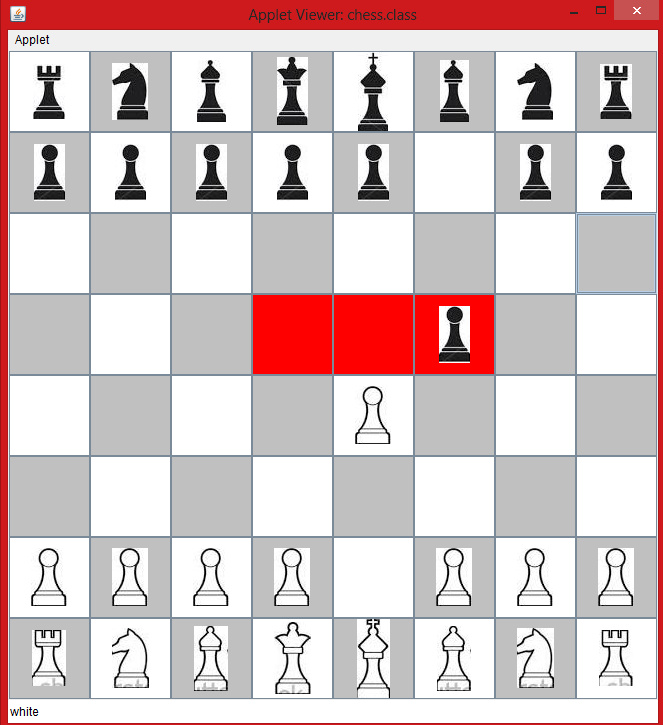
Function pawnpro() is scope handling function for pawn testing. Following things happen inside pawnpro() function: -

* Row and column no. of clicked button is extracted from variables mentioned above.
* Then it is checked whether piece is black or white. If counter is set to 1 i.e. chance is of white and user tries to click on black pawn, then nothing will work.
* If user clicks on a white pawn, then there is a for loop from col -1 to col+1 on position row+1 or row -1. Row and col are variables extracted by button pressed in first step. This completes the first phase of move. And scope is defined for pawn.
* It is also checked if any one of these four parameters is going out of bounds: row+1,col+1,col-1,col or row-1,col+1,col-1,col, if any of these variable’s value is going out of chessboard bounds then that value would b discarded.
* If in between scope marking some piece is discovered as obstruction, then scope formulation loop will break. It will be checked if obstruction piece is user’s or opponent’s, if the piece is of user then it won’t be marked inside scope, if it is of opponents it’ll be marked.

Further in programming of piece moving it has been programmed that if first phase is executed by a pawn then, only consider left and right column for cross occupation move i.e. if there is nothing at col-1 and col+1, do not move to that position, user can only move forward in that case.

**Overview**

It is quite clear that behavior of pawn is quite different in respect to other pieces. It is the only piece which needed extra programming for second phase of move. After simulation of pawn’s scope, I tested it on an empty chess board to make sure that in every condition, scope function is working fine for both black and white pieces.

****

**Queen**

**E:\workspaces\chess\queenb.jpg E:\workspaces\chess\queenw.jpg**

**Introduction**

The **queen** is the most powerful piece in the game of chess, able to move any number of squares vertically, horizontally, or diagonally. Each player starts the game with one queen, placed in the middle of the first rank next to the king. Because of the value of a queen, it is sometimes used as bait to lure an opponent into a trap by a queen sacrifice, as in a Fool's mate. Another tactic is to use the queen to threaten the opponent's queen, to either retreat or make a "queen trade" (losing both of them) to reduce the game to less-powerful pieces. The queen is often used in conjunction with another piece, such as teamed with a bishop or rook, where the pieces could guard each other while threatening the opponent pieces.

**Scope and movement**

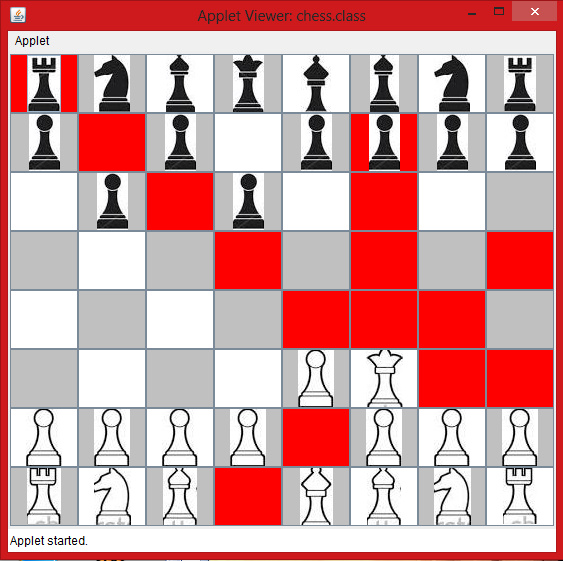
The queen can be moved any number of unoccupied squares in a straight line vertically, horizontally, or diagonally, thus combining the moves of the rook and bishop. The queen captures by occupying the square on which an enemy piece sits.

Although both players start with one queen each, a player can promote a pawn to any of several types of pieces, including a queen, when the pawn is moved to the player's furthest rank (the opponent's first rank). Such a queen created by promotion can be an additional queen, or if the player's queen has been captured, a replacement queen. Pawn promotion to a queen is colloquially called queening, which is by far the most common type of piece a pawn is promoted to because of the relative power of a queen.

**Implementation**

Implementation of queen required total of 8 for loops. Implementation of queenpro() is as follows: -

* First, the button which has been pressed is stored in variables of row and col.
* Two for loops are from 0 to col-1 and col+1 to 8.
* Two for loops are from 0 to row-1 and row+1 to 8.
* Four for loops of each diagonal pathways.
* If in between scope marking some piece is discovered as obstruction, then scope formulation loop will break. It will be checked if obstruction piece is user’s or opponent’s, if the piece is of user then it won’t be marked inside scope, if it is of opponents it’ll be marked.
* If user tries to click somewhere out of scope, everything will reset and user will have to start all over again from first phase.



**Knight**

**E:\workspaces\chess\knightb.jpg E:\workspaces\chess\knightw.jpg**

**Introduction and properties**

The **knight** is a piece in the game of chess, representing a knight (armored cavalry). It is normally represented by a horse's head and neck. Each player starts with two knights, who begin on the row closest to the player, one square from each corner. A knight is approximately equal in strength and value to a bishop. The bishop has longer range, but is restricted to only half the squares on the board. Since the knight can jump over pieces which obstruct other pieces, it is usually more valuable when the board is more crowded ([closed](http://en.wikipedia.org/wiki/List_of_chess_terms#Closed_game) positions). A knight is best when it has a 'support point' or [outpost](http://en.wikipedia.org/wiki/Outpost_(chess)) – a relatively sheltered square where it can be positioned to exert its strength remotely. On the fourth [rank](http://en.wikipedia.org/wiki/Rank_(chess)) a knight is comparable in power to a bishop, and on the fifth it is often superior to the bishop, and on the sixth rank it can be a decisive advantage. This is assuming the knight is taking part in the action; a knight on the sixth rank which is not doing anything useful is not a well-placed piece.

**Scope and movement**

The knight move is unusual among chess pieces. When it moves, it can move to a square that is two squares horizontally and one square vertically, or two squares vertically and one square horizontally. The complete move therefore looks like the letter *L*. Unlike all other standard chess pieces, the knight can 'jump over' all other pieces (of either color) to its destination square. It captures an enemy piece by replacing it on its square. The knight's ability to "jump over" other pieces means it tends to be at its most powerful in closed positions, in contrast to that of a bishop. The move is one of the longest-surviving moves in chess, having remained unchanged since before the seventh century. Because of this it also appears in most chess-related regional games. The knight moves alternately to light and dark squares.

A knight should always be close to where the action is, meaning it is best used on areas of the board where the opponent's pieces are clustered or close together. Pieces are generally more powerful if placed near the center of the board, but this is particularly true for a knight. A knight on the edge of the board attacks only three or four squares (depending on its exact location) and a knight in the corner only two. Moreover, it takes more moves for an uncentralized knight to switch operation to the opposite side of the board than an uncentralized bishop, rook, or queen. The knight is the only piece that can move at the beginning of the game without first moving a [pawn](http://en.wikipedia.org/wiki/Pawn_(chess)). For the reasons above, the best square for the initial move of each knight is usually one towards the center. Knights are usually brought into play slightly sooner than the bishops and much sooner than the rooks and the queen.

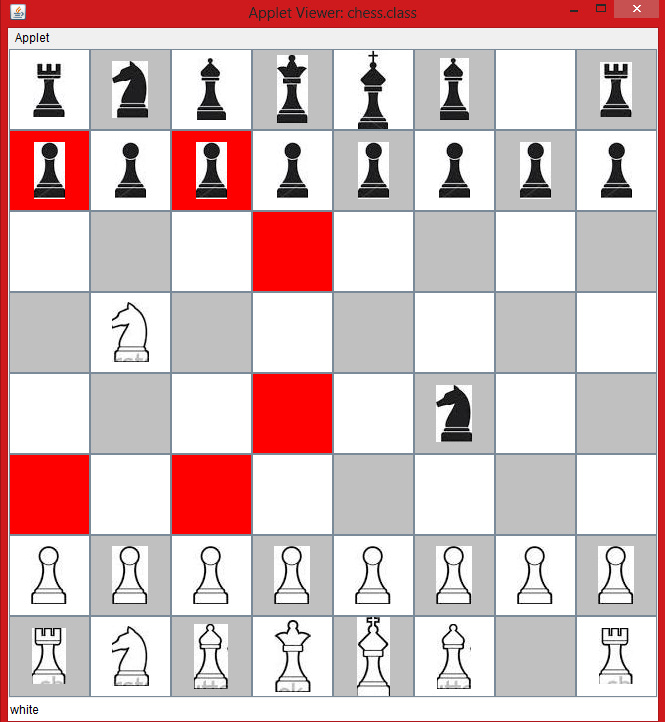
**Implementation**

Move of knight is being governed by function knightpro(). Following instruction execute in knightpro(): -

* First, the button which has been pressed is stored in variables of row and col.
* As scope of this piece has been explained above, all four position were marked regardless of obstructions.
* There are total 8 places where a knight could possibly move.
* Temporary variables containing value of present variables were assigned , that are tempr and tempc.
* At first increasing row by +1 and -1 and keeping col at col+2, gave first two locations.
* Then increasing row by +1 and -1 and keeping col at col-2, gave next two locations.
* Then increasing col by +1 and -1 and keeping col at row+2, gave next two locations.
* Then increasing col by +1 and -1 and keeping col at row-2, gave next two locations.
* If in between scope marking some piece is discovered as obstruction, then scope formulation loop will break. It will be checked if obstruction piece is user’s or opponent’s, if the piece is of user then it won’t be marked inside scope, if it is of opponents it’ll be marked.
* If user tries to click somewhere out of scope, everything will reset and user will have to start all over again from first phase.

**Obstructions and modifications**

As movement of knight is relatively different as compared to other chess pieces, I had to start from scratch. Initial concept of looping wasn’t effective anymore. There was also an advantage that all 8 positions of a knight are fixed. So, all those 8 positions were targeted in knightpro() with a simple concept that if any of these 8 positions go out of bound or is targeting any piece of same color as of user, than that position is to be discarded. A knight is most effective when placed in a weakness in the opponent's [pawn structure](http://en.wikipedia.org/wiki/Pawn_structure), i.e. a square which cannot be attacked by enemy pawns, so strength of a pawn is quite important for game.



Following image shows move of a white knight

**KING**

E:\workspaces\chess\kingw.jpgE:\workspaces\chess\kingb.jpg

**Introduction and properties**

The **king** is the most important piece. The object of the game is to trap the opponent's king so that its escape is not possible (checkmate). If a player's king is threatened with capture, it is said to be *in check*, and the player must remove the threat of capture on the next move. If this cannot be done, the king is said to be in checkmate. Although the king is the most important piece, it is usually the weakest piece in the game until a later phase, the endgame.

In conjunction with a rook, the king may make a special move called castling, in which the king moves two squares toward one of its rooks and then the rook is placed on the square over which the king crossed. Castling is allowed only when neither the king nor the castling rook has previously moved, when no squares between them are occupied, when the king is not in check, and when the king will not move across or end its movement on a square that is under enemy attack. This move is not demonstrated in this project.

In the opening and middle game, the king will rarely play an active role in the development of an offensive or defensive position. Instead, a player will normally try to castle and seek safety on the edge of the board behind friendly pawns. In the endgame, however, the king emerges to play an active role as an offensive piece as well as assisting in the promotion of their remaining pawns.

**Stalemate**

A stalemate occurs when, for the player with the move: -

* Player has no legal move.
* King is not on check.

If this happens, the king is said to have been stalemated and the game ends in a [draw](http://en.wikipedia.org/wiki/Draw_(chess)). A player who has very little or no chance of winning will try to incite opponent to inadvertently place the player's king in stalemate in order to avoid a loss.

**Scope and movement**

A king can move one square in any direction (horizontally, vertically, or diagonally) unless the square is already occupied by a friendly piece or the move would place the king in check. As a result, the opposing kings may never occupy adjacent squares, but the king can give discovered check by unmasking a bishop, rook, or queen. The king is also involved in the special move of castling.

**Implementation**

Implementation of king is simulated using kingpro() function: -

* First, the button which has been pressed is stored in variables of row and col.
* Temporary variables containing value of present variables were assigned , that are tempr and tempc.
* A for loop from tempc-1 and tempc+1 to simulate forward/backward scope on tempr+1.
* other for loop from tempc-1 and tempc+1 to simulate forward/backward scope on tempr-1.
* Individual selection of left and right adjacent squares next to king.
* If in between scope marking some piece is discovered as obstruction, then scope formulation loop will break. It will be checked if obstruction piece is user’s or opponent’s, if the piece is of user then it won’t be marked inside scope, if it is of opponents it’ll be marked.
* If user tries to click somewhere out of scope, everything will reset and user will have to start all over again from first phase.

**Obstruction and modifications**

In early phases of development, including check property in kingpro() seemed to be a fine idea, but in later stages it became hectic because of strange implementation of check programming function and its pre-calling on each move. So, check function is implemented separately. Implementation of check function has been explained in further stages. After simulation of king’s scope, I tested it on an empty chess board to make sure that in every condition, scope function is working fine for both black and white pieces in normal cases as well as in check conditions.

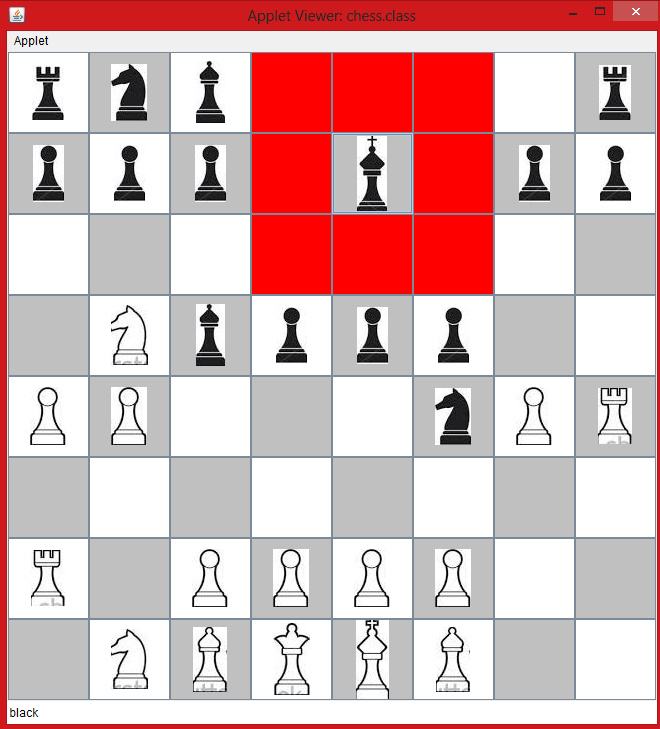


Image of scope of a white king

**ROOK**

E:\workspaces\chess\rookb.jpgE:\workspaces\chess\rookw.jpg

**Introduction**

A rook is a piece in the strategy board game of chess. Formerly the piece was called the *tower*, *marquess*, *rector*, and *comes* . The term *castle* is considered informal, incorrect, or old-fashioned.[[1]](http://en.wikipedia.org/wiki/Rook_(chess)#cite_note-1)[[2]](http://en.wikipedia.org/wiki/Rook_(chess)#cite_note-2)

Each player starts the game with two rooks, one in each of the corner squares on his own side of the board.

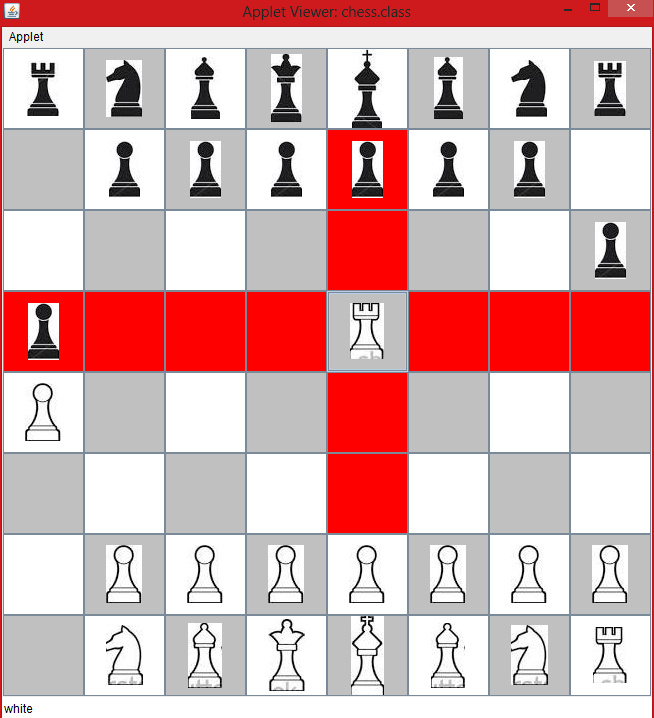
**Scope and movement**

The Rook moves horizontally and vertically any number of squares, forwards or backwards. There is one more special move one could do involving the rook called castling, it has been explained in castling section of King.

**Implementation**

Implementation of rook is handled using rookpro(). Following instructions are used:-

* First, the button which has been pressed is stored in variables of row and col.
* Temporary variables containing value of present variables were assigned, that are orow and ocol.
* 2 for loop had been used marking rows from 0 to row-1 and row+1 to 8.
* Other 2 for loop had been used marking rows from 0 to col-1 and col+1 to 8.
* If in between scope marking some piece is discovered as obstruction, then scope formulation loop will break. It will be checked if obstruction piece is user’s or opponent’s, if the piece is of user then it won’t be marked inside scope, if it is of opponents it’ll be marked.
* If user tries to click somewhere out of scope, everything will reset and user will have to start all over again from first phase.



**Bishop**

**Introduction**

A **bishop** is a piece in the board game of chess. Each player begins the game with two bishops. One starts between the king's knight and the king, the other between the queen's knight and the queen. In algebraic notation the starting squares are c1 and f1 for White's bishops, and c8 and f8 for Black's bishops.

**Scope and movement**

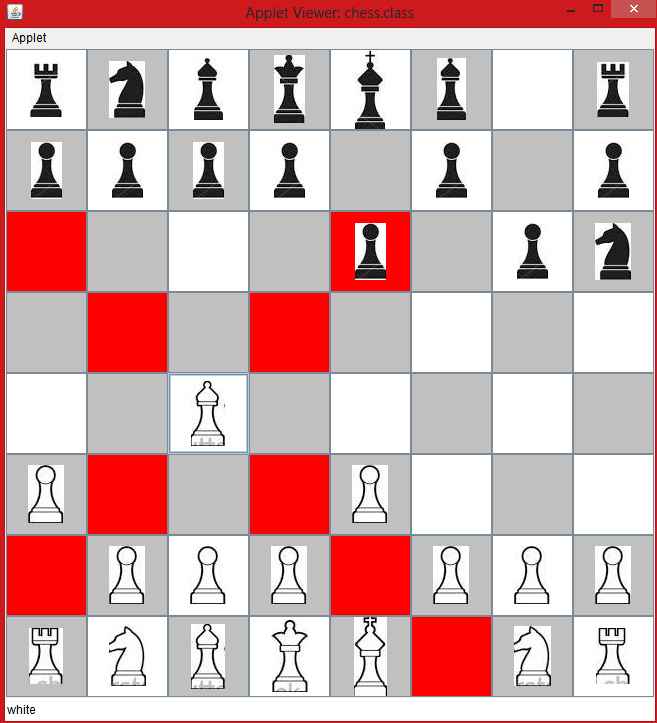
The bishop has no restrictions in distance for each move, but is limited to diagonal movement. Bishops, like all other pieces except the knight, cannot jump over other pieces. A bishop captures by occupying the square on which an enemy piece sits.

As a consequence of its diagonal movement, each bishop always remains on either the white or black squares, and so it is also common to refer to them as *light-squared* or *dark-squared* bishops.

**Implementation**

Implementation of bishop is governed by bishoppro method. Instructions assigned to this function are as follows: -

* First, the button which has been pressed is stored in variables of row and col.
* Temporary variables containing value of present variables were assigned, that are orow and ocol.
* 2 for loop had been used marking squares on first diagonal, one for upper diagonal part and one for lower diagonal part.
* Other 2 for loop had been used marking squares on second diagonal, one for upper diagonal and one for lower diagonal.
* If in between scope marking some piece is discovered as obstruction, then scope formulation loop will break. It will be checked if obstruction piece is user’s or opponent’s, if the piece is of user then it won’t be marked inside scope, if it is of opponents it’ll be marked.
* If user tries to click somewhere out of scope, everything will reset and user will have to start all over again from first phase.



**CHECK PROGRAMMING**

**Introduction**

**Check**

A **check** is a condition in chess that occurs when a player's king is under threat of capture on their opponent's next turn. A king so threatened is said to be *in check*. If the player cannot move out of check, the game ends in checkmate and the player loses.

If a player's move places the opponent's king under attack, that king is said to be *in*[*check*](http://en.wikipedia.org/wiki/Check_(chess)), and the player in check is required to immediately remedy the situation. There are three possible methods to remove the king from check:

* On moving the king to an adjacent non-threatened square, a king *cannot* castle to get out of check.
* Interposing a piece between the king in check and the attacking piece in order to break the line of threat (not possible when the attacking piece is a knight, or when in double check).
* Capturing the attacking piece (not possible in double check, unless the king captures)

If none of these three options are possible, the player's king has been *checkmated* and the player loses the game.

**Types of check**

A simple and very common type of check is when a piece the opposing king moves to directly attack only by itself. Sometimes such a check is part of a chess tactic such as a fork, a skewer, or a discovered attack on another piece. In some cases, a check can be used to defend against such tactics.

There are also a few more special types of check:

* *Discovered check*. A discovered check is similar to any other type of discovered attack except that it is a discovered attack on the opposing king. In a discovered check, a piece moves out of the line of attack by another piece so that this other piece (which can be a queen, rook, or bishop) is then checking the opponent's king. The piece that actually moved in the discovered check move could possibly be any type of piece belonging to the same player as the checking piece except queen or the same type of piece administering the check. A discovered check could be a tactic in itself because the piece that moved could attack or otherwise create a threat to another piece on the checked king's side. The opponent has to get out of the discovered check on the following move and may not get a chance to thwart the attack by the other piece that moved.
* *Double check*. A double check is a check from two pieces to the opponent's king in a single move. This happens when a moved piece attacks the king, resulting in a second piece giving check by discovered check. It can also happen, though very rarely, when an *en passant* capture opens two lines of attack simultaneously. In algebraic chess notation, a double check move is sometimes noted with a "**++**" after the written move in place of the usual "**+**", although "**++**" has been used to indicate checkmate (along with "**#**").A double check cannot be blocked, nor can it be met by capturing one of the checking pieces (unless the king itself makes the capture), because there is check from two directions.
* *Cross-check*. When a check is answered by a check, particularly when this second check is delivered by a piece blocking the first, it is called a *cross-check*. In fact, a "cross-checkmate" is also possible in that way (that is, to answer a check with a checkmate) but since no such term exists, it would be called cross-check as well. Cross-checks are rather rare but are a popular theme in chess problems.

Further I want to add that in this project the concept that I have used can simulate all three of them, but it can fail in few case too, for example, take double check, it could be perfectly simulated using check programming, both kings would get their background as black, but checkmate for both of them is not programmed.

**Implementation**

Implementation of chess function is illustrated bellow:-

* A temporary variable for turn is created in which current actual turn is saved.
* Then a for loop from 0 to 63, for checking each piece at each square, the loop starts from square zero checking for pieces at panel no. 0,1,2,3,…,63, if the piece found on that square is same as that of given in that temp variable then scope of that piece will be called.
* So at the end of the loop, scope of every piece will be marked. Whether it is white or black.
* Then it is checked for both the kings if they are under scope of opponent’s pieces to attack.
* If they are not, then function returns normally and painterpro() function is called which paints the whole board and opponent’s turn starts.
* If any one of the king is in direct scope of any piece of chess board, then it checks for color of the king piece. For sake of example let’s say, it is black, a variable kingcheckb will be set to 1 giving warning to opponents that the king is on check. the king which is on check, is painted by yellow colour in its background. This is the first part of check programming.
* On other subsequent calling of check function, if kingcheckb or kingcheckw is set to 1, and after analyzing the scope of every piece, the king is once again in scope of some piece then the game is over for that player. All buttons are deactivated and Joptionpane dialog box is flashes with message of who won.
* If on other subsequent calling of check function, if kingcheckb or kingcheckw is set to 1, and after analyzing the scope of every piece, the king is not in scope of some other piece, then the kingcheckb or kingcheckw variable is again set to 0.
* At the end of the function, temporary turn variable’s value is again transferred to initial turn variable.

**Obstructions and modifications**

Programming this particular function was a bit of a challenge as this function is very important for this game and simulation of real check is a real challenge. I want to make it clear that I haven’t simulated real check programming but the very concept of check. If the king gets under the scope of opponents piece after two subsequent moves, then the game is over. Initially, there was a huge mess inside my head about how to simulate check and then I got this idea of checking the scope of both white and black pieces. After every move, scope of all the pieces is checked by turn and then board is repainted leaving the king’s button if it is on check. Complexity of this function is huge, but because of given time in project and complex structure of project, this is all I was able to simulate. Rest assured, check function is working fine for all normal cases of game.

**Miscellaneous functions**

**Painter ()**

**Description**

This is a small utility function in this project; nonetheless it has a huge importance in this project. Work of this small utility function is painting the chess board; by paint it means set the correct background color for each button after the move is finished. It is also readily used in check programming. Button can be painted using setBackground() method.

**Implementation**

Implementation of painter function is rather simple. Instructions inside this method are as follows: -

* A loop that counts from 0 to 63, signifying the position of each button.
* On every count of loop, break it in row and column variables.
* If button’s background is yellow, and kingcheck variable is set to 1, there is no need to paint that button, so continue.
* If loop count starts at new even row, start from white and paint all the way back to the end by black and white color alternatively.
* If loop count starts at new odd row, start from black and paint all the way back to the end by black and white color alternatively.

**Sysexit ()**

**Description**

This is also a small utility function; it has been made exclusively for check function. When checkmate conditions arise, this function is called; it disables all buttons for further usage.

**Implementation**

Instructions inside this method are: -

* Call to painter to function, that clears the chess board before shutting it down.
* Memory counter is incremented, to make this Sysexit changes irreversible (see memory retention chapter) .
* A for loop from 0 to 63, which is to get control of each button. On each count of for loop corresponding button is disabled using setEnable() method.

**MEMORY RETENTION**

**Description**

This is an additional feature that I added in this project. Memory retention, being more precise is actually undo move. Undo Move is a function inside a chess program to update the internal [chess position](https://chessprogramming.wikispaces.com/Chess+Position) and its [Board representation](https://chessprogramming.wikispaces.com/Board+Representation) as well as associated or dependent state variables and data by a move unmade on the internal board. In undo move, reversible aspects of a position can be [incrementally updated](https://chessprogramming.wikispaces.com/Incremental+Updates) by the [inverse](http://en.wikipedia.org/wiki/Inverse_function) or [own inverse](http://en.wikipedia.org/wiki/Involution_%28mathematics%29) operation of [Make Move](https://chessprogramming.wikispaces.com/Make+Move). Irreversible aspects of a position, such as [up state](https://chessprogramming.wikispaces.com/En+passant), [castling rights](https://chessprogramming.wikispaces.com/Castling+rights) and the [half move clock](https://chessprogramming.wikispaces.com/Halfmove+Clock) are either restored from a [stack](https://chessprogramming.wikispaces.com/Stack)([LIFO](http://en.wikipedia.org/wiki/LIFO_%28computing%29)), or simply kept in arrays indexed by current search or game [ply](https://chessprogramming.wikispaces.com/Ply). Alternatively, one may capacitate the move with all the necessary information to recover those irreversible aspects of a position as well. As castling and en peasant moves are not implemented in this following project, they are out of picture. In following project, I have made memory retention capacity of just one move i.e. only last move can be undone.

**Implementation**

A whole new class of memory.java is created for memory retention purpose. This class is simply the container of last function. In beginning of each move, all data values for e.g. last piece selected or moved, its color, row number, col number, name of the piece, etc. are stored in temp variables. It has been implemented in program using following logic: -

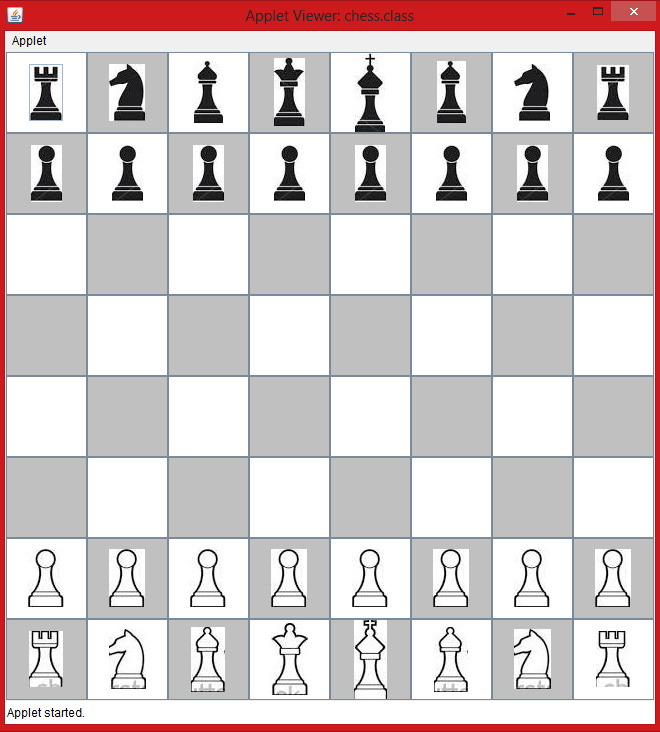
* A nested key listener is used inside action performed method. ‘z’ key of keyboard is bonded with key listener i.e. as soon as ‘z’ key is pressed, key listener fires itself.
* Inside key pressed function, there is a for loop which checks each button, if none of them is red. Then main function won’t be executed. A static ‘z’ variable is incremented on each red button encounter. This step signifies that undo function should only be called when second phase of move is over.
* The pressed key is extracted using getKeyCode method, and pressed key is checked against z.
* Counter variable in memory class is used as exclusion variable. It ensures only one move is reversible.
* There is an unusual problem with key listeners, it has been discussed below, but for tackling that problem, a mutual exclusion variable ‘a’ is used.
* If z is 0, memory counter is zero, a is 0, and pressed button is z, then main function will be executed.
* In main function, all previous steps are deleted and states of all temp variables are staged.

**Obstructions and modifications**

There is an unusual problem with key listeners, It haunted me for a long time but thanks to internet I found a solution. When a key is pressed and there is a key listener to listen to that event, four to five times listener function is fired on single key press. To tackle this problem I used mutual exclusion variable ‘a’, on each function call ‘a’ is incremented by 1, this inhibits the execution of function again. After execution of function is over, On next calling, ‘a’ variable is refreshed and initiated to zero.

**Playing Chess**

After the GUI window appears upon execution, the ﬁrst player (white chess pieces) clicks on the piece that (s)he would like to move and then clicks on a valid position for this piece (including capturing another piece, which then disappears from the board). Invalid moves are not allowed. A player is also not allowed to move another player’s pieces. Alternate chances will be given to both black and white side. The background color yellow under king’s grid square tells the player when (s)he is in check, and the player must move the King to get out of check. If after player’s move, the background under king’s grid square is still yellow, the game is over. I have played many games of Chess, and in a few rare instances, an exception occurred. The majority of the time, however, this Chess game worked well. Figure below is a screen shot of the Chess game.



**Summary**

In this paper I presented the classic version of Chess implemented as a GUI. I successfully created a game using Java swing applet and swing components. Although I encountered problems using swing framework, I was able to design a working, user-friendly graphical interface for this Chess game.

**References**

[1] Herbert Schlidt complete reference JAVA 2009 Swing chapter

[2] Wikipedia.com

[3] Stackoverflow.com