CHECKERS
CS 101
FINAL PROJECT
POST – MORTEM

For: Professor Hao Jiang

By: Patrick W. Murphy

May 8th 2009
**Project Overview:**

My goal for the final project was to use the Java programming language to create a fully functional “Checkers” game i.e. one that follows all of the standard rules of Checkers, with a simple, intuitive, graphical-user-interface.

**Programming Style:**

It was my original intention to develop the game using an Object-Oriented Programming method, in which the primary objects would have been the board, each of the pieces, a scoreboard, and a “listener” object, which would interface with the board and the pieces to enforce the rules of the game.

Due to difficulties I experienced while attempting to create such self-maintaining, functional objects, I wound up writing the program with only three classes: (1) the primary class, “Checkers.java” which contains the `main()` method, which calls all of the other classes / objects / methods, (2) the class “Board.java”, which handles the initialization of the graphical representation of the checkerboard and the executes calls to the `drawPiece()` method to place all the pieces on the board in their starting position, and (3) the class “Listener.java”, which contains all of the dynamic drawing methods, as well as all of the conditional structures that implement the rules of the game.

While the primary user-interface is graphical, there is verbose text output via the terminal. Though especially useful for debugging purposes, this terminal output can also be redirected to a text-file for later review (e.g. if the players want to review the specific moves they made in a game, keep track of how many of their pieces were “kinged,” etc.

Within the `main()` method of Checkers.java, two global, static objects are created, one of type “Board” and the other of type Listener. It is especially important that these objects are global so that the code within each of the other two classes can access and modify the variables and execute methods associated with the objects – most importantly, the variables and methods associated with the listener.
object; it is the listener object which contains the array that numerically represents the state of each square on the checkerboard and this array is written to from both the Board class and from within the Listener class itself.

**Methods by Class and their Function(s):**

1. Checkers.java
   a. Main(String args[])
      i. Takes first two strings in the args[] and marks them as player names.
      ii. Calls drawBoard() and initialBoardSetup() via the board object.
      iii. Calls listen() via the listener object – also passes player names to the object.

2. Board.java
   a. drawBoard()
      i. Uses StdDraw library to create a graphical representation a checkerboard.
   b. initialBoardSetup()
      i. Draws all pieces in their initial positions using the drawPiece() method and assigns appropriate numerical values to the 8x8 2D array board[][], which represents the status of each square – empty or occupied by a certain type of playing piece - numerically. Both the method drawPiece() and board[][] array are accessed from the board class via the global listener object.

3. Listener.java
   a. Listen()
      i. The overarching method which encapsulates the drawing methods and logical structures which govern movement, jumping, and score keeping within the game.
   b. drawPiece()
      i. Handles the drawing of pieces according to player number, which determines color, and type of piece (regular or king) which also modifies piece color. Player 1, regular = round piece with a red color. Player 1, king = round piece with a green color. Player 2, regular = round piece with a grey color. Player 2, king = round piece with a blue color. Any time a piece is drawn this method automatically modifies the status of that square in the board[][] array accordingly (0 = empty, 1 = has a regular red piece, 2 = has a regular grey piece, 3 = has a green king piece, 4 = has a blue king piece.)
   c. clearPiece()
      i. Handles the elimination of pieces from the board due to their being “jumped” by an opposing piece. Pieces are eliminated visually by painting over them using StdDraw, and the board[][] array is modified to reflect the new empty state of the square from which the piece was eliminated i.e. that square’s position in the array is reset to 0.
   d. movePiece()
      i. “Moves” pieces using a combination of drawPiece() and clearPiece() according to logical structures within the method which represent the rules of the game.
   e. keepScore()
i. Increments int playerScore_1 and int playerScore_2 accordingly each time that player jumps an opposing player’s piece, then outputs the new value of the current player’s score to the terminal.

**Current Functionality:**

In its current state, the game implements a partially functional graphical user interface, which represents a 2D checkerboard of the appropriate dimensions with the appropriate number of pieces. Pieces for each player are initialized as regular in type. Player 1 always begins the game, and turns alternate thereafter between player 1 and player 2 after the completion of one move by the current player (i.e. selection of a piece, selection of a new square for that piece). Precise data about the location of the piece selected, and current player number, and current click in the two-click cycle are output to the terminal window.

Players who are moving one of their regular pieces may “jump” one of the other player’s regular pieces in any diagonal direction so long as it would not result in the movement of their piece backwards on the board (relative to the opposing player’s side of the board). Each time an opposing player’s piece is jumped, the current player’s score is increased by 1 and the current score of that player is output to the terminal.

If the current player moves one of his or her pieces into a square in the last row opposite his or her side of the board, then his or her piece will change color (depending on player number) to signify that this piece has now become a “king” piece. Also, output in the terminal will note that the current player has been “kinged”. This “kinged” piece will now be able to move both forwards and backwards on the board and its status as a king will be marked accordingly in the board[][] array.

If at some point one of the player’s pieces are entirely eliminated from the board, the opposing player wins. This is noted in the terminal as soon as the next player tries to take his or her turn.
**Functionality Only Partially Implemented or yet to be Implemented:**

I underestimated the amount of time it would take me to develop a fully functional version of the game. As such, functions that I either had trouble implementing, or simply did not have time to implement, but which would need to be implemented in order for the game to be considered fully functional, are listed and described below:

1. **Completed logic for “jumping” using king type pieces.**

   The logic which governs jumping other pieces using king type pieces is incomplete. At present, jumping using a king piece will only work for (1) player 1 king pieces (green) jumping a regular player 2 piece diagonally such that where the original location of the king piece is \((x,y)\) the final location is equal to \((x-2, y+2)\) or \((x+2,y+2)\).

   Attempting to jump using a player 2 king piece (blue) either result in movement of the piece without execution of the jump, or depending on location, an arrayOutOfBounds error. Most likely this error results from a conditionals within the logic sequence that attempt to check the value of an array index that does not exist (i.e. the variable which denotes the index is calculated to be less than 0, which is an impossible value for an array index.

2. **Completed logic to constantly evaluate for all “win” scenarios.**

   Currently the logic governing testing for a winner will only recognize a scenario in which all of a player’s pieces have been eliminated by the opposing player. Logic which will check for a scenario in which no moves are possible still needs to be developed. This would involve counting through the board[][] array continually while checking to see if it is the case that the current player cannot make a legal move, if so, the opposing player automatically wins.
3. Logic to allow for multiple jumps by one player by blocking the playerId switch until multiple jump is completed.

   This logic remains to be implemented. Admittedly this would be easier to implement had I structured the turn switching to be based on user input (i.e. the user presses a button to indicate he or she has completed all desired (legal) moves and the game changes the playerId accordingly).

4. Logic to prevent players from moving pieces on to white squares.

   This would be fairly easy to implement with a conditional construction. At present, pieces may be moved on to white squares, but attempting to jump a piece from a white square, or move to another white square will cause problems as it will attempt to “clear” the old white square resulting in it being painted over by a filled black square, ruining the board representation.
1. GUI (BOARD REPRESENTATION)

Cell status at 1,0 is equal to 0
Cell status at 1,0 is equal to 1
Cell status at 3,0 is equal to 0
Cell status at 3,0 is equal to 1
Cell status at 5,0 is equal to 0
Cell status at 5,0 is equal to 1
Cell status at 7,0 is equal to 0
Cell status at 7,0 is equal to 1
Cell status at 9,1 is equal to 0
Cell status at 9,1 is equal to 1
Cell status at 2,1 is equal to 0
Cell status at 2,1 is equal to 1
Cell status at 4,1 is equal to 0
Cell status at 4,1 is equal to 1
Cell status at 6,1 is equal to 0
Cell status at 6,1 is equal to 1
Cell status at 1,2 is equal to 0
Cell status at 1,2 is equal to 1
Cell status at 3,2 is equal to 0
Cell status at 3,2 is equal to 1
Cell status at 5,2 is equal to 0

Begin setup of player 1 and player 2 pieces in their initial positions...
2. RED PIECE MOVED

3. GREY PIECE MOVED
pressed 2 playerId = 2
For click 2 values are: 3 4 and player id is: 2
Cell at 4,5 was cleared.
Cell status at 4 5 is equal to 0
Cell status at, 3,4 is equal to 0
Cell status at, 3,4 is equal to 2

4. GREEN KING PIECE
5. BLUE KING PIECE

```
For click 1 values are: 3 6 and player id is 1
pressed 2 playerId = 1
For click 2 values are: 4 7 and player id is: 1
Cell at 3,6 was cleared.
Cell status at 3,6 is equal to 0
Cell status at, 4,7 is equal to 0
Cell status at, 4,7 is equal to 3
Player 1 was kinged!
```

```
pressed 2 playerId = 2
For click 2 values are: 5 0 and player id is: 2
Cell at 4,1 was cleared.
Cell status at 4,1 is equal to 0
Cell status at, 5,0 is equal to 0
Cell status at, 5,0 is equal to 4
Player 2 was kinged!
```

6. PLAYER 2 WIN: (Player 2’s name = Leo. Entered as command line arg)
Player 2's score is: 12

pressed 1 playerId = 1
Congratulations, Leo you've won the game!
For click 1 values are: 2 1 and playerId is 1