Class: main

Package: Main

Description: Main class initializes object components of the game and connects them together.

Interface

Uses: Gameboard, Menu, Handler, saveMenu, loadMenu

Access Programs: None

Implementation

Uses:

java.io.File;
java.io.FileNotFoundException;
java.io.PrintWriter;
java.io.UnsupportedEncodingException;

Variables:

menu: Menu
game: Game
load: LoadMenu
handler: Handler
save: saveMenu

Access Programs: None

Local Programs: None

Class: Menu

Package: UI

Description: Menu is the main menu of the game where games can be started and loaded from, and settings can be changed.

Interface

Uses: Handler

Access Programs:

pic(): void: changes image to still picture.

red(): void: changes image to red piece jumping in animation.

grey(): void: changes image to grey piece jumping in animation.

music(): JButton: returns local variable sound.

clip(): AudioClip: returns local variable clip.

start(): JButton: returns local variable start.

Single(): JRadioButton: returns local variable Single.

Double(): JRadioButton: returns local variable Double.

load(): JRadioButton: returns local variable load.

Implementation

Uses:

java.applet.Applet;
java.applet.AudioClip;
java.net.URL;
java.awt.*;
javax.swing.*;
javax.swing.border.Border;

Variables:

sound: JButton
panel: JPanel
url: URL
clip: AudioClip

Single: JRadioButton
Double: JRadioButton
start: JButton
load: JButton
picture: JButton
pic: Icon
red: Icon
grey: Icon
startPic: Icon
loadPic: Icon
loweredBorder: Border

Access Programs:

pic(): void: sets the main menu image to the Icon pic.

red(): void: sets the main menu image to the Icon red.

grey(): void: sets the main menu image to the Icon grey.

music(): JButton: returns local variable sound.

clip(): AudioClip: returns local variable clip.

start(): JButton: returns local variable start.

Single(): JRadioButton: returns local variable Single.

Double(): JRadioButton: returns local variable Double.

load(): JRadioButton: returns local variable load.

Local Programs:

none

Class: gameboard

Package: UI

Description: Main class initializes object components of the game and connects them together.

Interface

Uses: Handler

Access Programs:

Player(): int: returns the number of the active player.

setPlayer(): void: allows us to set the player whose turn it currently is.

SwitchPlayer: JButton: returns the JButton switchPlayer.

ToMenu(): JButton: returns the JButton toMenu.

Init(): void: sets up a regular game

customInit(): void: sets up the initial board without the piece for a custom game.

getColour(): Icon: returns the state or the colour of the square.

SetupComplete(): JButton: returns the JButton setupComplete.

SetlblPlayer(): void: will show users whose turn at the moment.

board(): int[][]: will return the current board.

board(int board[][]) : void: will set the input board to the current board).

Implementation

Uses:

java.applet.Applet;
java.applet.AudioClip;
java.awt.BorderLayout;
java.awt.Dimension;
java.awt.FlowLayout;
java.beans.PropertyChangeEvent;
java.beans.PropertyChangeListener;
java.net.URL;
javax.swing.*;
javax.swing.border.Border;

Controllers.Handler;

Variables:

```
private int[][] board = new int[8][8];
        private int player;
        private URL url = Menu.class.getResource("lose.wav");
        private URL url1 = Menu.class.getResource("C6.wav");
        private URL url2 = Menu.class.getResource("C7.wav");
        private URL url3 = Menu.class.getResource("C5.wav");
        private URL url4 = Menu.class.getResource("E5.wav");
        private URL url5 = Menu.class.getResource("E6.wav");
        private URL url6 = Menu.class.getResource("Roll.wav");
        private AudioClip lose = Applet.newAudioClip(url);
        private AudioClip C6 = Applet.newAudioClip(url1);
        private AudioClip C7 = Applet.newAudioClip(url2);
        private AudioClip C5 = Applet.newAudioClip(url3);
        private AudioClip E5 = Applet.newAudioClip(url4);
        private AudioClip E6 = Applet.newAudioClip(url5);
        private AudioClip Roll = Applet.newAudioClip(url6);
        private JButton music;
        private Timer timer;
        private JLabel message;
        private JLabel lblplayer;
        private JButton toMenu;
        private JButton setupComplete;
        private JButton saveGame;
        private JPanel panel1;
        private JPanel panel2;
        private JPanel panel3;
        private JPanel panel4;
        private JButton switchPlayer;
        private JButton btn00,btn01,btn02,btn03,btn04,btn05,btn06,btn07;
        private JButton btn10,btn11,btn12,btn13,btn14,btn15,btn16,btn17;
        private JButton btn20,btn21,btn22,btn23,btn24,btn25,btn26,btn27;
        private JButton btn30,btn31,btn32,btn33,btn34,btn35,btn36,btn37;
        private JButton btn40,btn41,btn42,btn43,btn44,btn45,btn46,btn47;
        private JButton btn50,btn51,btn52,btn53,btn54,btn55,btn56,btn57;
        private JButton btn60,btn61,btn62,btn63,btn64,btn65,btn66,btn67;
        private JButton btn70,btn71,btn72,btn73,btn74,btn75,btn76,btn77;
        final int[] whitesquares = {0,2,4,6,9,11,13,15};
        final int[] brownsquares = {1,3,5,7,8,10,12,14};
        final private Border loweredBorder = BorderFactory.createLoweredBevelBorder();
        final private Icon white =  new ImageIcon(getClass().getResource("white.png"));
        final private Icon blackselected = new ImageIcon(getClass().getResource("selected.png"));
        final private Icon brown = new ImageIcon(getClass().getResource("brown.png"));
        final private Icon black = new ImageIcon(getClass().getResource("black-brown.png"));
        final private Icon red = new ImageIcon(getClass().getResource("red-brown.png"));
        final private Icon redselected = new ImageIcon(getClass().getResource("redselected.png"));
```

```
final private Icon blackking = new ImageIcon(getClass().getResource("black-brown-
king.png"));
final private Icon redking= new ImageIcon(getClass().getResource("red-brown-king.png"));
final private Icon blackkingselected = new ImageIcon(getClass().getResource("black-brown-
king-selected.png"));
final private Icon redkingselected = new ImageIcon(getClass().getResource("red-brown-king-
selected.png"));
final private Icon[] squarecolour = new Icon[BTNS.length];
```

Access Programs:

Player(): int: returns the number of the active player.

setPlayer(): void: allows us to set the player whose turn it currently is.

SwitchPlayer: JButton: returns the JButton switchPlayer.

ToMenu(): JButton: returns the JButton toMenu.

Init(): void: sets up a regular game

customInit(): void: sets up the initial board without the piece for a custom game.

getColour(): Icon: returns the state or the colour of the square.

SetupComplete(): JButton: returns the JButton setupComplete.

SetlblPlayer(): void: will show users whose turn at the moment.

board(): int[][]: will return the current board.

board(int board[][]) : void: will set the input board to the current board).

Local Programs:

none

Class: saveMenu

Package: UI

Description: saveMenu class allows the user to save games.

Interface

Uses: Handler

Access Programs:

save: JButton: returns ok

getSaveName: String: returns the saved game name.

setNotification(String not): void: sets the notification to the string given in the parameters.

clearTextField: void: clears the text field of all characters.

cancel: JButton: returns cancel JButton.

Implementation

Uses:

javax.swing.JButton;
javax.swing.JFrame;
javax.swing.JTextArea;
javax.swing.JTextField;

Variables:

gameName: JTextField
ok: JButton
cancel: JButton
notification: JTextArea

Access Programs:

save: JButton: returns ok

getSaveName: String: returns gameName.getText();

setNotification(String not): void: runs notification.setText(not)

clearTextField: void: game.setText(“”)

cancel: JButton: returns cancel.

Local Programs:

none

Class: LoadMenu

Package: UI

Description: Load class allows user to load previously saved games.

Interface

Uses: Handler

Access Programs:

Implementation

Uses:

```
java.awt.BorderLayout;
java.awt.Dimension;
java.awt.FlowLayout;
java.awt.event.ActionListener;
java.awt.event.ActionEvent;
javax.swing.*;
javax.swing.border.Border;
```

Variables:

scrollPane: JScrollPane

Load: JButton

toMenu: JButton

panelTop: JPanel

panelBottom: JPanel

panelWhole: Jpanel

instr: JLabel

savedGames: String[]

Access Programs:

Load: JButton: returns load.

Loadlist: void: creates new JList list, `scrollPane.setViewportView(list)`

`getSavedGames: String[]: returns savedGames`

```
setMessage(message: String): void: instr.setText(message)
```

Local Programs:

none

Class: Handler

Package: Controllers

Description: Handler class handles the consequences that result from a user clicking a GUI component.

Interface

Uses: Checkers, main

Access Programs:

CustomSetup():void: sets up the board for a custom setup.

CheckersSquareClicked(int board[][], int i):void: handles the consequences of clicking a given square.

setupCompleteClicked():void: ends custom setup mode.

SwitchPlayerClicked():void: switches players, or in the case that setup is complete, becomes the Resign button, allowing players to resign.

displaySetupMessage():void: sets the initial message the player sees.

LoadGameClicked():void: Loads the selected saved game into the board.

SaveGameClicked():void: Saves the current game under the name the user inputs.

LoadButtonClicked():void: Opens the load menu.

aiMove:void: Makes the computer make a move for the active player.

AIcomment:void: Displays a commentary on the current state of the game.

checkGameOver:void: Ends the game if the game is over.

Implementation

Uses:

```
java.awt.event.ActionListener;
java.awt.event.ActionEvent;
java.io.BufferedReader;
java.io.BufferedWriter;
java.io.File;
java.io.FileNotFoundException;
java.io.FileReader;
java.io.FileWriter;
java.io.IOException;
java.io.PrintWriter;
```

```
java.io.UnsupportedEncodingException;
java.util.Arrays;
java.util.List;
java.util.Random;
javax.swing.*;
```

Variables:

gameOver: boolean
animationCounter: int
gameSound: boolean
themeCounter: int
setup: int
blacks: int
reds: int
message: String
menu: Menu
game: Gameboard
load: LoadMenu
save: saveMenu
row: int
col: int
board: int[][]
player: int
singlePlayer: int
computerPlayer: int
difficulty: int

Access Programs:

CustomSetup():void: is a variation of CheckersSquareClicked() (described below) because CustomSetup() indicates what clicking on a square does while there are pieces (reds or blacks) that still need to be placed before the game can begin. Every time a black pieces is placed in a valid position blacks=blacks-1 will run (same thing goes for reds). However, if the user clicks on an invalid square, no counter will be reduced, and instead the global variable “message”, which is displayed on the gameboard, will tell the user that they have tried to place a piece on an illegal square.

CheckersSquareClicked(int board[][], int i):void: handles all of the logic behind the consequences of a user pressing a specific square given a specific game state. This method takes in the i-th button and then converts this into coordinates by using row=i/8 and col=i%8. The row and col variables represent the row and col of the square that was just clicked. From there, it activates one of several cases which will determine the change that is applied to the board. Uses a switch statement to deal with the various cases.

setupCompleteClicked():void: will activate when the user clicks the JButton “setupComplete”, which the user can use during custom setup to indicate that they are done setting up and that the game can begin. This will be used whenever 12 black pieces or 12 red pieces are not needed (for instance if the user is setting up a game partway through). For this reason, the method will set global variables reds=0

and blacks=0, which will cause the CustomSetup() method to no longer be called when squares are clicked.

SwitchPlayerClicked():void: will activate when the user clicks the JButton "switchPlayer". This method has two if-statements that will change the value of game.player to 1 (using encapsulation method game.setPlayer(1)) if the current value of player is 2, and likewise, it will change the value from 2 to 1 if the current player value is 2.

displaySetupMessage():void: sets the initial message the player sees.

LoadGameClicked():void:opens up a new frame where a list of games can be viewed and a specific game can be selected. That game data is then read and placed into the game board array and the checkers board is then updated and displayed. Inside this method, the saveData.txt file (which is the game's save data) is checked for existence. If it exists, then the user can select from the list view which game they would like to load. Each game is given a game name during the save process.

SaveGameClicked():void: handles if "save" is clicked in the game window. If the file "saveData.txt" does not exist, it is created in the current directory. If it already exists, it continues on with the save procedure. During this "save" procedure, a text box is opened up in which the user can specify what they would like to name the game. The game name is the first element in the comma-separated line of data. After the game name is written to saveData.txt, the data of the array is appended to the same row of data in the file. This method also prints the location of the saveData.txt file.

LoadButtonClicked():void: handles when the "load" button is clicked. It handles the actual loading of the data in saveData.txt. The method searches for the specified game name and loads that game's game board array, or data, through a buffered reader and then writes it to the game's array, and sets up the new game board.

aiMove:void: uses inhereted methods from the Checkers class in order to choose a move to make. Then it will select the initial square and move it to the final square using local functions. A more detailed description can be found on page 16 of this document.

AIcomment:void: uses a random number to select one of 31 cases, each containing several comments that the AI can make, depending on the given state of the game.

checkGameOver:void: method is used after each move to see if the game is over. If the game is over, a sound clip is played according to who won, and the String message is updated with the results, namely "Player 1 wins!" or "Player 2 wins!", depending on who has more pieces when the game is done.

Class: Checkers

Package: Controllers

Description: Checkers class contains all of the methods for realizing checkers and it's rules.

Interface

Uses: None

Access Programs:

removeLegalsFor(int row, int col):void: removes legal moves for the given piece.

setLegalJumpsFor(int[][] board, int player, int row, int col):void: sets legal jumps for the given piece.

removeLegalJumpsFor(int,board[][], int row, int col):void: removes legal jumps for the given piece.

anyLegalJumps(int[][] board, int player):boolean: returns true if the given player can make a legal jump anywhere on the board, otherwise returns false.

canJump(int[][] board, int player, int row, int col):boolean: returns true if the given piece can jump, otherwise returns false.

jumpMove(int[][] board, int row, int col):void: handles the case where the selected piece is to make a jump.

getMoves(int[][] board, int player):ArrayList<int[][]>: returns a list of legal moves.

chooseMove(int[][] board, int player, int difficulty):int[][]: chooses a move from the legal moves based on the difficulty settings.

move(int[][] board, int player, int startRow, int startCol, int finishRow, int finishCol):void: moves piece at startRow, startCol to finishRow, finishCol.

ratio(int[][] board, int player):double: method returns the ratio of player pieces to enemy pieces.

getNumPieces(int[][] board, int player):int: method returns the number of pieces on the board belonging to player.

sumAllPieces(int[][] board):int: method returns the total number of pieces on the board.

copy(int[][] move):int[][]: makes a copy of the given 2D array.

isPiece(int[][] board, int player, int row, int col):boolean: returns true if the given square is a piece, otherwise returns false.

isLegal(int[][] board, int row, int col):boolean: method returns true if the given square correlates to a

legal move and false otherwise.
copy(int[][] board1, int[][] board2):void: method copies the values from board1 into board2, overwriting previous values in the process.

areEqual(int[][] board1, int[][] board2):boolean: returns true if the two arrays are equal, otherwise returns false.

Local Programs:

isLegal(int i):boolean: method returns true if the given number correlates to some legal move value and false otherwise.

getRatio(int[][] board, int playerTurn, int player, int counter):double: returns the ratio of player pieces to enemy pieces that will result from this board scenario.

setUpJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing):void:
sets the downward orientated jumps for the piece at row, col on the board.

maxIndex(ArrayList<Double> list):int: returns the index corresponding to the maximum ratio.

chooseRandom(int[][] board, int player):int[][]: returns a random legal move.

moveSelected(int[][] board, int player, int row, int col):void: moves the selected piece to the given coordinates.

minIndex(ArrayList<Double> list):int[][]: returns the index corresponding to the minimum ratio.

getSelected():int[]: returns the coordinates of the selected piece.

setSelected(int row, int col):void: sets the given piece to a selected piece.

setLegalsFor(int row, int col):void: sets the legal moves for the given piece.

Deselect():void: deselects any selected piece on the board.

regularMove(int[][] board, int row, int col):void: handles the case where the selected piece is to move in a regular fashion (ie. not a jump).

setDownJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing):void: sets the downward orientated jumps for the given piece.

Implementation

Uses:

```
java.util.ArrayList;
java.util.Random;
```

Variables:

none

Access Programs:

removeLegalsFor(int row, int col):void: will remove all the possible legal moves from the element board[row][col].
setLegalJumpsFor(int[][] board, int player, int row, int col):void: will set the legal jumps for the square given by the coordinates in the parameters. Uses setUpJumps and setDownJumps. setUpJumps will only run if the piece belongs to player 2 or it is a king. setDownJumps will only run if the piece belongs to player 1 or it is a king. The method checks if there are empty positions 2 diagonals away from the selected piece. If an enemy piece exists between the two then a legal jump is set. If a legal jump exists, all other legal moves are removed then the legal jumps are placed on the board.

removeLegalJumpsFor(int,board[][], int row, int col):void: will remove any legal jumps that could be made by the piece given by the coordinates in the parameters.

anyLegalJumps(int[][] board, int player):boolean: will return true if any piece on the board (belonging to "player") can make a jump. The method uses canJump method on each piece and returns true if any piece can jump, and false otherwise.

canJump(int[][] board, int player, int row, int col):boolean: will return true if the piece on the square given in the parameters can make any jumps. This method uses setLegalJumpsFor and removeLegalJumpsFor in a for loop. First it selects the given piece, then it sets the legals for the piece, and then searches the board for a jump, before remove legal jumps and deselecting the piece.

jumpMove(int[][] board, int row, int col):void: will handle the case where the selected piece is to move to the coordinates given in the parameters when this represents a jump. This involves removing the piece that is located between the selected piece's initial coordinates and final coordinates (given by row, col in the parameters).

getMoves(int[][] board, int player):ArrayList<int[][]>: iterates through the entire board, selecting each pieces that belongs to the player given in the parameters, and then checks to see if the piece can make any moves. If the piece can make a move, the move is added to an ArrayList by using a 2D int array as follows:

```
int[][] move = new int[2][2];
        move[0][0] = The Initial Row Coordinate
        move[0][1] = The Initial Column Coordinate
        move[1][0] = The Final Row Coordinate
        move[1][1] = The Initial Column Coordinate
```

Once every piece has been iterated through, the ArrayList containing all of the moves is returned.

chooseMove(int[][] board, int player, int difficulty):int[][]: first calls getMoves(board, player) in order to collect all the possible moves the player can make. Then, the method gets the ratio that each move will produce of player/enemy pieces using the getRatio method. Depending on the difficulty setting, the method will choose either the highest ratio or the lowest ratio. The difficulty setting Easy skips the ration process and chooses a move at random from the ArrayList using the method chooseRandom.

move(int[][] board, int player, int startRow, int startCol, int finishRow, int finishCol):void: method moves the piece at board[startRow][startCol] to the square board[finishRow][finishCol].

ratio(int[][] board, int player):double: method returns the ratio of player pieces to enemy pieces.

getNumPieces(int[][] board, int player):int: method sums the number of pieces on the board belonging to the player given by the parameters and then returns the value.

sumAllPieces(int[][] board):int: method adds the number of pieces of both players together and returns the value.

copy(int[][] move):int[][]: method iteratively copies each value from the given 2D array into a new 2D array which it then returns once the loop is complete.

isPiece(int[][] board, int player, int row, int col):boolean: method returns true if the number at board[row][col] belongs to the player given in the parameters. Otherwise it returns false.

isLegal(int[][] board, int row, int col):boolean: method returns true if the number given by board[row][col] correlates to a legal move and false otherwise.

copy(int[][] board1, int[][] board2):void: method copies the values from board1 into board2, overwriting previous values in the process, using an iterative for-loop.

areEqual(int[][] board1, int[][] board2):boolean: method iteratively compares each value of the two arrays to each other to see if they are equal, provided their index's are identical (e.g. board1[i][j]==board2[i][j]). If, for all values, the two boards are equal, then the value true is returned. If any one comparison is false, then the method returns false.

Local Programs:

isLegal(int i):boolean: method returns true if the given number correlates to some legal move value and false otherwise.

getRatio(int[][] board, int playerTurn, int player, int counter):double: is a recursive method that creates the recursion tree described on page 16. The base case is the method ratio(board,player) and the recursive step gets the average ratio of all the children using the getRatio method.

setUpJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing):void: sets the downward orientated jumps for the piece at board[row][col].

maxIndex(ArrayList<Double> list):int: method iterates through the list of ratios and returns the index of the list with the highest value ratio.

chooseRandom(int[][] board, int player):int[][]: method chooses a random move from the ArrayList created by getMoves(board,player). First the method creates a random number between 0 and the list size. Then it returns the move located at the random number's index.

moveSelected(int[][] board, int player, int row, int col):void: method is similar to the method in the Handler class, only it does not have some of the calls that the Handler method makes to game and

menu objects.

minIndex(ArrayList<Double> list):int[][]: method iterates through the array list and returns the maximum value (which corresponds to a ratio).

getSelected():void: uses two for loops to iterate through the entire 2D array (refer to Decomposition document for details on the 2D array). The method will search the array for a value which corresponds to a selected piece, and then it will return the selected piece. In the picture to the left the values {4,3} would be returned.

setSelected(int row, int col):void: increases the value of the element board[row][col] by 2, signifying that it is selected, and then calling the method setLegalsFor(row,col).

setLegalsFor(int row, int col):void: will set all the legal squares that the element board[row][col] can legally move to.

Deselect():void: will get the selected element from the array, and then decrease its value by 2, signifying that it is no longer selected, and then call the method removeLegalsFor(row,col).

regularMove(int[][] board, int row, int col):void: will handle the case where the selected piece is to move to the coordinates given in the parameters when this represents a regular move (from one square to an adjacent square).

setDownJumps(int[][]board, int player, int row, int col, int enemyPiece, int enemyKing):void: sets the upward orientated jumps for the piece at board[row][col].