Checkers

ASSIGNMENT 3- APRIL 13, 2014

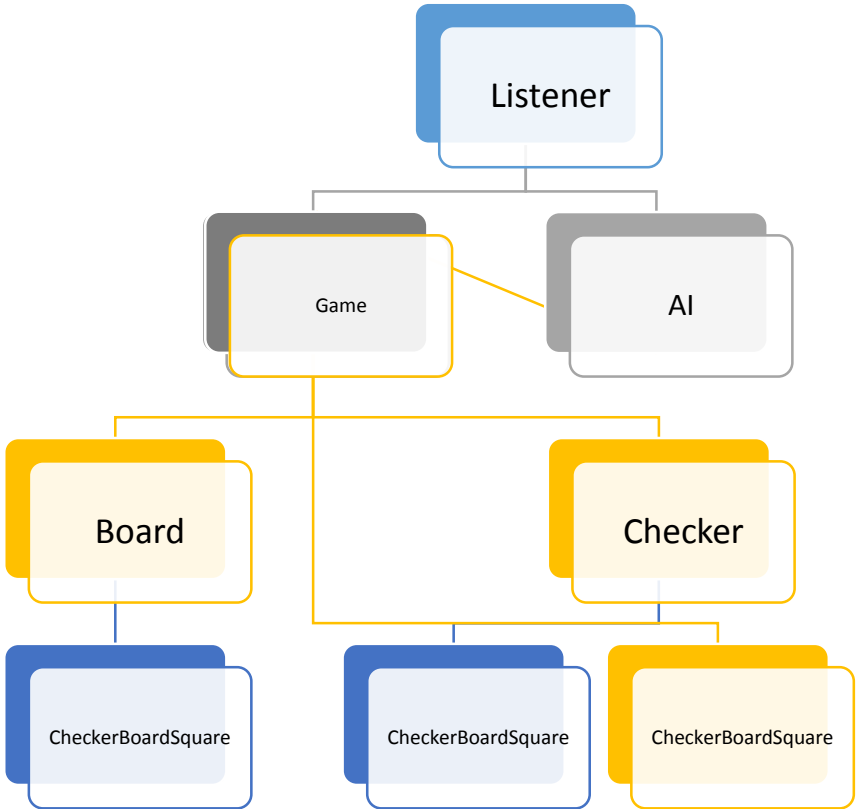
GROUP 18- HYUNA LOY, JENELL HOGG, KAREN DESA, MAHIMA PATEL, WARREN CORBEIL

Table of Contents

Introduction

Assignment 3 is the final version of the checkers game. It is a two player game; each player takes turns trying to gain an upper hand by taking as many pieces from the opposing player. In this assignment the user will be able to select whether they will play against the computer or another player. Additional to the functions from assignment 1 and 2, the user(s) will be able to make legal moves using the mouse. The computer player will have an algorithm that takes the weighing of the future moves and make the move that has the highest about of weighing. The following document will demonstrate how these requirements were met through the structure, methods, and testing of the code in Java.

Uses Relationship Diagram



Decomposition

The Listener Class acts as a controller. It creates and has access to all the views (Board class, CheckerBoardSquare class) and models(Checker class) through the game class. It also creates an AI class for single player mode. It receives user input and then passes it either to the AI class or the Game class to change the views and models depending on the state of the game.

The AI class contains methods that will identify what is the best move for the red checkers in single player mode given the current state of the game. Based upon its calculations, it will pass information to the Game class to make this best move.

The Game class acts as a model. It contains all the information about the game of checkers. The class receives inputs from the AI class or the Listener class and updates the models(Checkers) and views(Board, CheckerBoardSquare) accordingly.

The Board Class acts as a view. It is the graphical user interface.

The CheckerBoardSquare class helps construct the board class. Each instance of this class corresponds to a square on the checkerboard. This class contains methods to modify the display. (Place a checker on a square, remove a checker, etc.)

The Checker Class acts as a model . It gives the controller information about a checker piece so that it can properly update the board when a checker is moved. The controller can create an instance of class Checker when a checker is being placed for the first time. The Checker class also has access to an instance of the CheckerBoardSquare class so it can call methods to update the picture of a square when a checker is being placed upon it and update images when a checker becomes a king checker.

These classes have been chosen in order to ensure there is a good separation of concerns and to ensure that the uses relationship will be hierarchy. This means the design will be easy to create and maintain.

OVERVIEW

CLASS Listener

Sets up the game, AI and timer for the game to be fully initialized and calibrated for all game play modes.

INTERFACE

USES

NONE

TYPE

NONE

ACCESS PROGRAMS

Method Name	Type- Inputs	Input Info	Behaviour (Semantics)	Syntax Info
main	None		The main method, through which the whole program executes	Creates a new Listener
mouseClicked	MouseEvent	A click from the user On the G.U.I created by the Board Class.	Identifies where the user clicked and executes the appropriate methods to update the display and state variables accordingly.	Series of if and else if statements with function calls and updating of state variables nested within

mousePressed	MouseEvent	MouseEvent created when the mouse is being held by the user	Ensures all the abstract methods from the MouseListener Interface are present and that nothing changes when the mouse is being held.	This function is empty
mouseReleased	MouseEvent	MouseEvent created when the mouse is released by the user	Ensures all the abstract methods from the MouseListener Interface are present and that nothing changes when the mouse is released.	This function is empty
mouseEntered	MouseEvent	MouseEvent created when the cursor enters the listener	Ensures all the abstract methods from the MouseListener Interface are present and that nothing changes when the mouse enters the listener.	This function is empty
mouseExited	MouseEvent	MouseEvent created when the cursor exits the listener	Ensures all the abstract methods from the MouseListener Interface are present and that nothing changes when the mouse exits the listener.	This function is empty
Listener	None	None	Creates a new instance of the listener class. Sets up the initial models and views.	Creates a new instance of the board class. Adds a mouselistener to the mainWindow (JFrame) of the Board glass. Initializes the ArrayList squares and checkers.

IMPLEMENTATION

USES

```
import java.awt.Color;
```

VARIABLES

Variable Name	Type	Info
thisGame	Game	An instance of the Game class, used to create a game and direct mouse input to determine player turn state and timer delays for the ai.
Ai	AI	An instance of the AI class, calls the moveOnTurn method for the ai appropriately.
n	Listener	Creates an object of the Listener class
click	Point	Contains the point of a mouse event, deciphers it and passes it to be read.

ACCESS PROGRAMS

Method Name	Update info	Behaviour	Syntax info
decipherPoint		Using the point clicked, identifies where the user clicked on the GUI	The board is divided up into squares of dimensions 50x50 pixels. Taking the location of the clicked point in type double, it returns integers x and y containing the location closest to the point clicked location using Math.ceil(pointClicked). If the point clicked is on the bottom menu, it

		and returns a point corresponding to an area. Takes an exact point of a pixel and identifies what button or square on the board that pixel belongs to.	determines which menu button it is and returns a generalized coordinate for the menu item clicked. If the point clicked is not on the menu, it returns the integer location of the point clicked.

CLASS: GAME

Controls movement and rules of checker game and sets up game menu.

INTERFACE

USES

Listener

TYPE

NONE

ACCESS PROGRAMS

Method	**Output**	**Behaviour (Semantics)**
Public		
Game:	None	Creates a new checkerboard and checkers for the user.
readClicked(click: Point):	None	Finds the button or square and checker in the place that was clicked and determines the appropriate response. If a checker or square on the checkerboard was clicked, checks whether the square is a valid move, the square is invalid, or the square is the same one that was clicked and will either restrict or allow the user to make that move.

continueGame:	None	Uses the savedGame File from the state to restore a previously saved game.
moveChecker(moveToSquare: CheckerBoardSquare):	None	Given a square a selected checker can move to, moves it. If the selected checker captured another checker in doing so, removes the captured checker from the board and from checkers.
unselectAll:	None	Changes selectedChecker to null. Updates the images on the board so that any previously highlighted squares are now un-highlighted and any previously selected checkers appear unselected
validMoves(check: Checker):	None	Finds all the valid moves for a given checker and updates the ArrayList moves to contain the CheckerBoardSquares that correspond to these valid moves.
findChecker(square: CheckerBoardSquare): Checker	The checker on the square or null if there isn't	Looks through checkers to try and find a checker on the given square.
findValidCheckers:	None	Based on whose turn it is and what colour checker is selected, will make a list of valid checkers based on the list of captured checkers
colourCount: Point	A point in the format (x, y) where X corresponds to the number of white checkers on the board and y corresponds to the number of red	Iterates through checkers and counts all the red and white checkers separately.

	checkers on the board	
findSquare(x: Point): CheckerBoardSquare	A CheckerBoardSquare corresponding to the point given, or null if there isn't one	Iterates through the array squares to find a square to match the corresponding point. If the point is doesn't correspond to any of the squares on the checkerboard, this will return null.
Private		
saveGame:	None	Obtains all necessary information about the state of the system in order to restore that state and saves it to the savedGame file.
unhighLightSquares:	None	None
checkIfKinged(square: CheckerBoardSquare, check: Checker):	None	Identifies whether a checker should be Kinged when placed on a square, and if so, kings the checker and modifies the CheckerBoardSquare to hold a king checker of the same colour.
customGameSetup:	None	Changes the GUI and the state variables so that the user is able to start setting up a custom game.
highlightPossibleMoves:	None	Highlights the possible squares a selected checker can move to
unselectSquares:	None	Deselects the currently selected square.
canBeCaptured(possibleSquare: CheckerBoardSquare , currentSquare:	The square a checker can move to. This may or may not be a capturing move. It will return null if	Evaluates whether a checker on the currentSquare can make either a regular move to the possibleSquare or a capturing move past the possibleSquare. Returns the square the checker on the currentSquare would move to, or null if cannot move.

CheckerBoardSquare): CheckerBoardSquare	the Checker on the currentSquare cannot capture a checker found on the possibleSquare	
startGame:	None	Sets up the board and updates state variables to show the game has started in two player mode
putChecker(square: CheckerBoardSquare, isRed: Boolean):	None	Creates a new checker instance and places it upon the given CheckerBoardSquare Updates the arrayList checkers to include the new checker for later access by the Listener Class. Will also delete a checker on the square if there is already one placed.
removeChecker(square: CheckerBoardSquare):	None	Removes a checker visually from the board. Removes the corresponding instance of the Checker Class from checkers.
colourCount: Point	A point in the format (x, y) where X corresponds to the number of white checkers on the board and y corresponds to the number of red checkers on the board	Iterates through checkers and counts all the red and white checkers separately.
reset:	None	Clears all checkers from the board and restores original menu
standardGameSet	None	Creates new instances of Checker with appropriate field variables assigned and updates the

up:		Board to show a game of checkers in standard layout (White Checkers on the bottom, red on the top)

IMPLEMENTATION

USES

java.awt.Color;
import java.awt.Point;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

VARIABLES

Variable Name	**Type**	**Info**
checkers	ArrayList<Checker>	Holds all instances of the class Checker that represent checkers on the board
checkerBoard	Board	All the graphical elements that make up display of the checkerboard
squares	Array<CheckerBoardSquare>	Holds all the instances of the CheckerBoardSquare that represent squares on the board

selectedChecker	Checker	Represents an instance of the Checker class that a user has selected
moves	ArrayList<CheckerBoardSquare>	Holds all instances of the CheckerBoardSquare class that correspond to valid places selectedChecker can move to.
captureSquares	ArrayList<CheckerBoardSquare>	Holds all the instances of the CheckerBoardSquare class that correspond to squares on the board containing checkers that could be captured by selectedChecker
isCustomGame	Boolean	Is true when a user selects a custom game and is in custom game setup. Is false otherwise
isGameStarted	Boolean	Is true when the game has started.
toPlaceRed	Boolean	Is true when a user indicates they wish to place red checkers during custom game setup. Switches to false when the user indicates they wish to stop putting red checkers down.
toPlaceWhite	Boolean	Is true when a user indicates they wish to place white checkers during custom game setup. Switches to false when the user indicates they wish to stop putting white checkers down.
validCheckers	ArrayList<Checkers>	Stores a list of valid checkers for the user
player1Turn	boolean	Is true when it is player 1’s turn
singlePlayerMode	Boolean	Is true when single player mode is selected
isThereACaptureMove	boolean	Is true when it is possible to make a capture move against opponent
savedGame	File	A file which is written to and read from when the user indicates they wish to save or continue a game.

ACCESS PROGRAMS

Method	Input Info	Update info	Syntax info
Public			
Game:	None	None	Creates a new Board and new checkers
readClicked(click: Point):	None	toPlaceRed toPlaceWhite player1Turn	Through a series of if statements, finds the place that the user clicked (i.e. square, checker, button) using the point location and state variables. The point location determines which side of the menu was clicked. Further if statements determine which button on the specified side of the menu was clicked and an appropriate response is given. If a checker is selected, if statements are used to determine whether the square is a valid move, invalid move, the square is the same on previously clicked or in a position to be kinged. An appropriate response is then given.
unhighLightSquares:	None	None	This function is empty.
unselectSquares:	None	None	Calls function unHighLightSqaures
findSquare(x: Point): CheckerBoardSquare	Point location of a mouse click on the main window of the game.	None	Given a point, uses a for loop to find a square with the same point in it. Returns the square containing specified point or null if nothing is found.
continueGame:	None	Modifies squares, checkers, moves, isGameStarted, selectedChecker, checkerBoard	Uses an ObjectInputStream to recover the Object array of state data that was put into the file savedGame by the method saveGame. The state information is restored by reassigning the state variables to the restored values The method setUp() from the Board class is then called to restore the display of the saved game. This method will print an error message to the user if no game has been saved previously.
moveChecker(moveToSquare: CheckerBoardSquare):	A square on the board that a selected checker will move to		Using a series of if statements, may or may not execute the method moveTo from the Checker class to move a checker and removeChecker from the Listener class to remove a captured checker.

unselectAll:	None	selectedChecker CheckerBoardSquares found within moves	Calls the methods unhighlight from the CheckerBoardSquare class to un-highlight any previously highlighted squares (from the ArrayList moves). Calls upon the method restoreImageInPlace from the Checker class to remove the image of a selected Checker.
validMoves(check: Checker):	Any given checker	Empties moves	First uses standardMoves from the Checker class to get possible moves for the checker. Checks to see if the squares relating to the ArrayList of Point objects obtained from standardMoves have checkers on them. (These squares are found by using the method findSquare from the Listener class) The method findChecker is called to identify whether any of the squares related to the previously mentioned squares have checkers on them. Finally, ithe canBeCaptured method from the Listener class is called to identify whether any of the checkers can be captured and if so, returns the square where the Checker check would move to if it captured said checker. If the checker has no valid moves, the ArrayList moves has nothing added to it and is of size zero.
findValidCheckers:	None	None	Creates two ArrayList<Checker> to store valid checkers and checkers that can be captured. It iterates through all checkers, if it is player 1's turn and the checker is not red or if it is not player 1's turn and the checker is red, it will add the checker either to validCheckers or captureCheckers. The remaining if statements alter validCheckers values based on the size of validCheckers and captureCheckers. If captureCheckers is not empty, validCheckers evaluates to captureCheckers, if validCheckers is not empty, validCheckers remains the same, else validCheckers is null. A new array list is made for moves and captureSquares.
Private			
saveGame:	None	Modifies savedGame	Creates an array of Objects of the state variables moves and squares. Also gets information from the checkers ArrayList about the instances of

			checker on the board (where they each are, what colour they are, and whether they are a king or not) and saves the information in the array of Objects as well. The array of Objects is then saved into the savedGame file by using an ObjectOutputStream.
checkIfKinged(square: CheckerBoardSquare, check: Checker):	CheckerBoardSquare - a square that the checker is placed on or will be placed on Checker- Said checker	Modifies the Checker and CheckerBoardSquare given	Using a series of if statements and state data, decides whether the method becomeKing from Checker class is to be executed.
customGameSetup:	None	isCustomGame checkerBoard	Executes customGameMenu from the Board class to update the menu on the GUI. Then sets isCustomGame to true.
highlightPossibleMoves:	None	The CheckerBoardSquares found within moves	Executes highlightSquare from the CheckerBoardSquare class on the appropriate squares. These are always the squares from the ArrayList moves.
canBeCaptured(possibleSquare: CheckerBoardSquare , currentSquare: CheckerBoardSquare): CheckerBoardSquare	possibleSquare- the square a checker could move to that may or may not have a checker on it currentSquare - the square a checker is currently placed	None	Using the method findChecker, identifies whether the possibleSquare has a checker on it. Method accesses the field data in the possibleSquare and currentSquare. (See CheckerBoardSquare class's public entities formore info) Using the field location, the method compares the location of the possibleSquare and currentSquare to analyze whether the checker would move up and to the left, up and to the right, down and to the left, or down and to the right in order to capture a checker on the possibleSquare. Knowing what direction the checker would move in, the method uses findSquare to access the square that the checker would move to if it captured. If findSquare returns null or a square with a checker on it, the square cannot be captured. If the square cannot be captured, this method returns null, if not, it will

			return a square the checker can move to.
findChecker(square: CheckerBoardSquare): Checker	Square- any given square	None	Has a for loop that iterates through the ArrayList checkers, trying to find a checker that has a place (See public entities in the Checker Class for more info) that matches the square given.
startGame:	None	isGameStarted isCustomGame checkerBoard	Sets toPlaceRed and toPlaceWhite to false to prevent the next mouse click from producing a checker. Through a series of if statements, it assesses the state of the board i.e. the number of checkers and acts accordingly. If there are no checkers on the board and the game is custom, it outputs a message via the method setMessage(Board) to place more checkers on the board. If there are only checkers of one color on the board, it prompts the user via setMessage(Board) to place a checker of another color on the board. Otherwise, it starts the game by set setting isGameStarted to true and updating the menu via the method startGameMenu(Board). If the game is not custom, it sets the game up in standard layout via standardGameSetup(), else sets isCustomGame to false and the game is started.
putChecker(square: CheckerBoardSquare, isRed: Boolean):	Square- any square on the board isRed- corresponds to whether the checker to be put down is red or not	The input square Checkers- to contain a new instance of the checker class and may possible remove some	Through a series of if-else statements, checks what the user selected checker color is and prepares the required checker image. Using method hasChecker(square), it determines if there is a checker on the selected square. If so, it removes the checker from the board via removeChecker(square) and the checker array and replaces it with the newly user selected checker via placeCheckerOn(checkerColour). An instance of checker is created for the new checker. Using method checkIfKinged(square,check), the new instance of checker's status is checked based on its location, if it is in a position to be kinged, it's status is changed to King. The instance of checker is added to the checkers array.
removeChecker(square:	Square- any square on the board	The input square Checkers	Removes checker from board via method call takeCheckerOff(). Using a for loop and if statement, finds the instance of the removed checker and

CheckerBoardSquare):	containing a checker		removes it from the checker array.
colourCount: Point	None	None	A for loop iterating through the ArrayList checkers with a condition that recognizes what colour the checker is and counts it as either a red or white checker. Once the for loop terminates, the method is able to construct a Point with the form previously mentioned in the return info column.
findSquare(x: Point): CheckerBoardSquare	A point that may or may not correspond to a square on the board	None	Using a for loop and if statements, if the array of squares is not empty , it checks if the location given is inside any of the checkerboard squares. Otherwise, it returns null
reset:	None	isCustomGame toPlaceRed toPlaceWhite isGameStarted selectedChecker checkers	
standardGameSetup:	None	Squares Checkers CheckerBoard	Using two for loops, one for red and white checkers, creates instances of red/white checkers and places red/white checkers on checker squares that are black. The new instances of checker are then added to checkers array.

CLASS: CheckerBoardSquare

INTERFACE

USES

Listener

TYPE

NONE

ACCESS PROGRAMS

Method Name	**Output**	**Behaviour**
Public		
CheckerBoardSquare(m: Point):	None	Stores location into a variable m.
placeCheckerOn (checkerImage: String):	None	Creates a new imageIcon with the image related to the String given and adds it to the JLabel checkerPosition, which is added to the CheckerBoard square and painted on
takeCheckerOff:	None	Updates the CheckerBoardSquare to display no checker. Also updates the state (hasChecker to false)

highlightSquare:	None	Creates a new imageIcon with the highlighted square image and displays it on the CheckerBoardSquare
unhighlightSquare:	None	Removes the image of a highlighted square off the checkerBoardSquare
Private		
createImageIcon (path: String): ImageIcon	An image icon with an image file corresponding to the String given. If there is no file, returns null.	Identifies if an image Icon can be made given a file path and returns an image icon if possible.

IMPLEMENTATION

USES

```
java.awt.BorderLayout;
import java.awt.Point;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JPanel;
```

VARIABLES

Variable Name	Type	Info
Location	Point	A point of the form (column, row) that corresponds to a location on the checkerboard this CheckerBoardSquare is displayed
hasChecker	Boolean	Indicates if the CheckerBoardSquare has a checker on it (True if it does, false if not)
checkerPosition	JLabel	This JLabel is what will hold the images of checkers on the CheckerBoardSquare
glowSquarePosition	JLabel	This JLabel is what will hold the image of a highlighted square on the CheckerBoardSquare

ACCESS PROGRAMS

Method Name	Input Info	Updates	Syntax Info
CheckerBoardSquare(m: Point):	indicates position of each checker board square	None	Constructs a location variable m.
placeCheckerOn (checkerImage: String):	checkerImage- the name of the image file corresponding to a picture of	checkerPosition hasChecker=true	Sets hasChecker to true, uses removeAll() to clear the square of previous checkers. Instantiates checkerPic (by calling method createImageIcon), an instance of ImageIcon and then adds it checkerPosition. Sets the layout of the instance to a BorderLayout and then adds checkerPosition to the centre. Repaints and revalidates the

	a checker		instance of the object.
takeCheckerOff:	None	hasChecker=false	Sets hasChecker to false, uses the remove method inherited from JPanel to remove checkerPosition. Then revalidates and repaints itself.
highlightSquare:	None	glowSquarePosition	Instantiates glowSquare (by calling on createImageIcon) and glowSquarePosition. Sets the layout of the instance of CheckerBoardSquare to a border layout and adds glowSquarePosition to itself. Then revalidates and repaints.
unhighlightSquare:	None	CheckerBoardSquare	Uses remove inherited from the JPanel to remove glowSquarePosition. Repaints the instance of itself.
Private			
createImageIcon (path: String): ImageIcon	path – states the location of the image file.	None	Loads the image from the URL specified in path and stores it in imgURL. If imgURL is not empty, returns a new ImageIcon, else prints that it couldn't find the path and returns null.

CLASS: CHECKER

Handles all checker attributes including a tally of the number/position of each checker, whether some are kinged or not, the pictures of them, the feature of them being highlighted and the movement of each checker.

INTERFACE

USES

Listener

TYPE

NONE

ACCESS PROGRAMS

Method Name	Input	Input Info	Return	Return Info
standardMoves	None	None	ArrayList<Point>	An ArrayList of Point objects in the format (column, row) that correlates to the squares that a checker could possibly move to. Note: These are not necessarily valid moves.
becomeKing	None	None	None	None
select	None	None	None	None
restoreImageInPlace	None	None	None	None
moveTo	CheckerBoardSquare Square	Square- where the checker will move to	None	None

IMPLEMENTATION

USES

```
import java.awt.Point;
import java.util.ArrayList;
```

VARIABLES

Variable Name	Type	Info
isRed	Boolean	Indicates whether the checker is red or not
Place	CheckerBoardSquare	The CheckerBoardSquare corresponding to the place on the board where the checker is located
isKing	Boolean	Indicates whether the checker is a king or not

ACCESS PROGRAMS

Method Name	Updates	Functions called	Behaviour	Syntax info
standardMoves	None	None	Identifies and returns what squares a checker could possibly move to based upon the state of the checker (What colour it is, where it is currently placed, whether or not it is a king)	The value of isKing is made true. Then using a if-else statement, a red checker piece places the RedKingC image and same for if the checker piece is black.
becomeKing	isKing=true place	placeCheckerOn (CheckerBoardSquare)	Transforms the checker into a King checker. Updates the checkerBoardSquare to display a picture of a king checker of the same colour	The value of isKing is made true. Then using a if-else statement, a red checker piece places the RedKingC image and same for if the checker piece is black.

select	Place	placeCheckerOn (CheckerBoardSquare)	Updates the image of the checker to appear selected	Using if-else statements the method checks if the selection is red or black and king or regular and depending on this a image will be placed on the board.
restoreImageInPlace	Place	placeCheckerOn (CheckerBoardSquare)	Restores the image of a checker in its place if it has been unselected or moved	After checking what kind of check piece is selected the place method is used to place image on the board.
moveTo	Place	takeCheckerOff (CheckerBoardSquare) restoreImageInPlace (Checker)	Visually removes a checker off of its place, then changes its place to a new CheckerBoardSquare. Then restores the image of the checker in its new place.	Place is given the value of the square from CheckerBoardSquare, and then the restoreImageInPlace is used.

CLASS: Board

Primarily responsible for the game's GUI; the board class initializes all menu buttons, their colours, the message bar and the checker square buttons that compose the playing field for the game.

INTERFACE

USES

LISTENER

TYPE

NONE

ACCESS PROGRAMS

Method Name	Input	Input Info	Return	Return Info	Update Info	Behaviour	Syntax Info
Board	None	None	None	None	All state info	Initializes the display. Creates squares- an array of CheckerBoardSquare instances that correspond to the squares on the board.	Sets up an 8x8 grid. Alternates the colours of the tiles on the grid by first using an if-else statement to alternate the colours of tiles the rows, and then another one to alternate the colours of the tiles of the columns. Adds the tiles to an array for later access. Instantiates all the JPanels in the menu, sets their colour and the text that is displayed and adds them to the menu. Then adds menu to mainWindow
setMessag	String	Message- a	None	None	messagePan	Sets the text of the	Uses the setText method

e	message	message to be displayed on the messagePanel on the display			el	messagePanel to display appropriate user prompts or error messages to the user. This method can also set the messagePanel to display no message if it previously was.	inherited from TextField to display a message on messagePanel
startGameMenu	None	None	None	None	menu	Removes all buttons from the menu, then adds the saveGame button, a blank panel, and a reset button. This is so the menu displays appropriate buttons to the user after the game is started.	Removes all the items from menu. Then creates a new blank Jpanel, sets its background colour to white, and adds saveGame, blank, and reset to menu and repaints
customGameMenu	None	None	None	None	menu	Removes all buttons from the menu, then adds the redButton, start, and whiteButton. This is so the menu displays the appropriate buttons to the user during a custom game set up	Uses the add method inherited from Jpanel to put redButton, start, and whiteButton to menu. Then revaldiates and repaints the view.
resetMenu	None	None	None	None	menu	Removes all buttons from the menu, then adds continueGame, start, and custom. This	Uses the removeAll method from JPanel to clear the menu. Then uses the add method to put continueGame, start and custom

						is so the menu displays the appropriate buttons to the user after the reset button has been clicked.	buttons onto the menu and revalidates and repaint.
setUp	An array of CheckerBoardSquare instances	This array relates to the checkerboard squares that would be displayed on a board	None	None	Pane	Given an array of CheckerBoardSquare, visually restores the board to hold these squares. Each square may or may not have a checker on them. This method is used to restore a saved game.	Calls startGameMenu, then removeAll from board. Uses a for loop to add all the squares to board, revalidates and repaints.
switchTurns	None	None	None	None	Player turn indicator	Checks the current colour of the turn indicator, upon call, switches this indicator to the colour of the other player to indicate the next player's turn	Toggles the background colour of the working tile from red to white or vice versa, and updates the text within that tile..

IMPLEMENTATION

USES

import java.awt.Color

VARIABLES

Variable Name	Type	Info
Menu	JPanel	Contains up to three menu buttons at a time
Reset	JPanel	A menu button. User clicks this when they wish to reset the board and restore the initial menu.
Start	JPanel	A menu button. User clicks this when they wish to start the game.
Custom	JPanel	A menu button. User clicks this when they wish to set up a custom game.
whiteButton	JPanel	A menu button. User clicks this when they wish to place white checkers on the board during custom game setup.
redButton	JPanel	A menu button. User clicks this when they wish to place red checkers on the board during custom game setup.
saveGame	JPanel	A menu button. User clicks this when they wish to save the current game.
continueGame	JPanel	A menu button. User clicks this when they wish to continue a previously saved game.
Board	JPanel	Contains all the CheckerBoardSquare instances corresponding to a Checker Board.
messagePanel	TextField	A panel that is used to display messages to the user.

Private Implementation

Variable Name	Type	Info

menu	JPanel	Contains up to three menu buttons at a time
reset	JPanel	A menu button. User clicks this when they wish to reset the board and restore the initial menu.
start	JPanel	A menu button. User clicks this when they wish to start the game.
custom	JPanel	A menu button. User clicks this when they wish to set up a custom game.
whiteButton	JPanel	A menu button. User clicks this when they wish to place white checkers on the board during custom game setup.
redButton	JPanel	A menu button. User clicks this when they wish to place red checkers on the board during custom game setup.
saveGame	JPanel	A menu button. User clicks this when they wish to save the current game.
continueGame	JPanel	A menu button. User clicks this when they wish to continue a previously saved game.
board	JPanel	Contains all the CheckerBoardSquare instances corresponding to a Checker Board
messagePanel	TextField	A panel that is used to display messages to the user.

CLASS: AI

INTERFACE

USES

Path

TYPE

NONE

ACCESS PROGRAMS

Method Name	**Output**	**Behaviour**
Public		
moveOnTurn	None	//AI evaluates the board and calculates all possible moves it can make. //The AI picks the most "valuable" or weighted move and moves there
public AI(Game thisGame)	None	Creates new Instance in Game

IMPLEMENTATION

USES

```
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import javax.swing.Timer;
```

VARIABLES

Variable Name	**Type**	**Info**
pathsOfAllCheckers	ArrayList of CheckerBoardSquares	Stores all the paths for every checker
valuesOfAllPaths	ArrayList	Stores the values corresponding to each path
paths	ArrayList	Stores all the paths for one checker
path	ArrayList	Stores a single path
moves	ArrayList	Initial Paths
markedSquares	ArrayList	Keeps tracks of the squares where the paths branch from
goneDownHere	ArrayList	Keeps track of the paths that have already been covered
validCheckers	ArrayList	Holds all the valid checkers
indexOfBestPath	Point	Stores the value of findIndexofMaxValue

checkerIndex	Int	The x-coordinate of indexOfBestPath
pathIndex	Int	The y-coordinate of indexOfBestPath
checkerToMove	Checker	Picks a checker from validCheckers based on the index specified in checkerIndex
pathToMove	ArrayList	Returns a square by using checkerIndex and pathIndex to select from pathsOfAllCheckers
maxChecker	Int	Initialized to zero
maxPath	Int	Initialized to zero
maxValue	Int	Initialized to zero
currentValue	Int	For iterating through valuesOfAllPaths in findIndexOfMaxValue
Values	ArrayList	Used in findSafetyOfAllPaths. Contains safety values of all paths
cloneOfValid	Checker	Cloned checker in findWeightOfAllCapturePaths
Source	CheckerBoardSquare	Holds the first value of path
valueOfMove	Int	Used to hold the weight of each move for findWeightOfCapturePath
possibleCaptureSquare	CheckerBoardSquare	Used to iterate through each each checker location
squareAroundPath	CheckerBoardSquare	Holds a square that is around a given path
Clone	Checker	Parameterized with finalSquare and true
placesAroundFinalSquare	ArrayList	Holds the location of all possible places around the final square
safetyOfPath	Int	Holds the safety of the path

ACCESS PROGRAMS

Method Name	**Input Info**	**Updates**	**Syntax Info**
moveOnTurn():	None	validCheckers	AI evaluates the board and calculates all possible moves that can be made by clearing paths, path, pathsOfAllCheckers, valuesOfAllPaths, and goneDownHere. Calls findValidCheckers from its Game instance and creates a new ArrayList to hold all the valid checkers. If there is a checker that can be captured, calls findAllCapturePaths for all validCheckers and finds their weight. Otherwise calls findAllNonCapturePaths and findSafetyOfPaths. Calls findIndexOfMaxValue to find the best path and then moves. Finally changes the value of player1Turn to true.
Move (indexofBestPath: Point, validCheckers: ArrayList<Checker>	Takes an ArrayList and an index of the best path in it		
revertPath():	None	None	If the path is not empty and at least one square has been marked, iterates through path and removes an item in path if it does not equal the last markedSquare
cloneChecker(c: Checker)	Takes in a checker	None	Creates a clone of the checker given to it and returns it
fineNonCapturePaths(clone : Checker)	Takes in a checker that has been	paths	Calls validMoves for the clone, clears current path. Iterates to add the place of the clone and moves to path. Then adds path to paths and clears path.

	cloned, not the original checker		
findAllNonCapturePaths(checkers: ArrayList<Checker>):	None	pathsOfAllCheckersPaths	Creates a clone checker. Goes through checkers and performs findNonCapturePaths with the clone.
findAllCapturePaths(checkers: ArrayList<Checker>):	None	pathsOfAllCheckers	Creates a clone Checker and goes through each checker in checkers to findCapturePaths of the clone. Then calls pathsOfAllCheckers to add paths.clone. Finally, clears the path
findCapturePaths(c: Checker):	None	None	Saves the current checker location. If there are multiple checkers, takes from location and marks current location. It moves to a valid location from the current location and adds it to goneDownHere. Repeats the process until there aren't any more moves and saves the whole set as a move path. Then goes back to the last marked square and does the same thing until there are no squares remaining
findSafetyOnSquare(finalSquare: CheckerBoardSquare, toMove: Checker):	None	None	Evaluates whether a red checker is safe from capture on a square. Returns 1 if safe and 0 if not.
findWeightOfCapturePath(path: ArrayList<CheckerBoardSquare>, toClone: Checker):	Takes in a checker that is to be cloned and a path	None	Gives weight to each path to make the algorithm intelligent. If the checker can take a king along its path it adds 4 to the weight. If the checker can take another checker, it adds 2. Returns the value of the captureMoves + the safety value
findIndexOfMaxValue	None	None	Returns a point in the form (Checker index, path index) that corresponds to the best move make of all checkers and all paths on the board. The checker index will let us

			find the validChecker with the best path from validCheckers. Path index is used to find the best path for a particular checker

Traceability

Requirement 1.1 Initially set up an 8-by-8 checkers board with dark and light squares. There must be a light square in the bottom right corner. The normal convention is to label the columns as A, B, …, H starting from the left, and the rows as 1, 2, …, 8 starting from the bottom.
This requirement is met in the Board class constructor.
When the board class is initialized, it sets up a series of JPanels and CheckerBoardSquare instances to create a graphical user interface. This includes the view of the initially empty checker board made of instances of the CheckerBoardSquare class.

Requirement 2.2 The user shall be able to set up an initial position of pieces on the board by specifying either the standard opening position (as shown in the figure), or by specifying positions for pieces using a notation such as A7=B (black piece on A7), or E3=W (white piece on E3), or F8=WK (white king on F8), etc. Users shall be warned if the position is illegal, and pieces must not be placed on illegal squares (white/light squares). A maximum of 12 white pieces and 12 black pieces may be placed on the board. Alternatively, pieces may be placed on the board using a graphical interface.
The initial display of the board shows the menu displaying 3 options to the user. "Continue Game", "Start Game", and "Custom Game".
If the user clicks the "Start Game" button on this menu, then the method standardGameSetup is executed in the Listener class. This creates a game with the standard layout described.
If the user clicks "Custom Game" the menu then changes the display to show the user different buttons; "To Place a Red Checker", "Start Game", and "To Place a White Checker."
The user indicates they want to start placing red checkers by clicking the button with "To Place a Red Checker" displayed on it. Upon doing this, a state variable in the Listener class, a Boolean named toPlaceRed, is switched to true.
The user can indicate they want to place white checkers on the board by clicking "To Place a White Checker", upon which a state variable in the Listener class, a Boolean named toPlaceWhite, is switched to true.
Upon the next click after selecting one of said checker buttons, the conditions in the mouseClicked method identify whether a valid square has been clicked and whether or not a checker should be placed depending on how many checkers of the same colour are on the board already. The method colourCount() helps fulfill this requirement.

Requirement 1.3 In the case that the standard opening position is not used, there shall be a way for the user to indicate that set up is complete.
The user indicates set up of a custom game is complete by clicking the "Start Game" button found on the menu. The series of conditions found within the mouseClicked button recognizes when this button is being pressed. Upon doing this, the Listener class will check to see if the game can be started, and if so, changes the appropriate state variables so that the code will recognize the game has been started.
In the starting menu, the user can click upon a button that reads "Continue Game" in the menu. Upon doing so, the mouseClicked() method recognizes that this button was pressed and executes the continueGame function.

Requirement 2.1 Start a game from a previously stored state (the state should be saved within a file – and you can decide whether you want to allow more than 1 saved game).
In the starting menu, the user can click upon a button that reads "Continue Game" in the menu. Upon doing so, the mouseClicked() method recognizes that this button was pressed and executes the continueGame function.

Requirement 2.3 Make moves – you don't need to be able to play a complete game (yet) – just move pieces from one position to another. The moves must be legal moves. You should be able to make moves that: simply move a piece to another square; jump the opponent's piece (so that piece is removed from the board); convert a piece to a "king"; move kings in both directions (forwards and backwards). You can decide how you want the user to indicate moves – graphically or by code (E3-D4 etc), or both.
The function moveTo moves pieces from one place to another.
When a user clicks a square on the board that contains a checker, the condition within the mouseClicked method in the Listener class recognizes this and stores the instance of the Checker class that represents the clicked checker in a state variable selectedChecker. The Listener class then executes the method validMoves which identifies all the valid moves for the selectedChecker and stores the instances of CheckerBoardSquare class that represent all the valid squares the selectedChecker could move to in the arrayList moves. Upon the next click, the conditions in the mouseClicked method will either move the selectedChecker to the clicked square if the clicked square corresponds to a valid move (is contained in the arrayList moves) or unselects the selected checker. This fulfills the requirement of being able to move a piece only when it is a legal move.
The methods checkIfKinged and becomeKing transform a checker into a king checker if the checker moves to a place where it must become a King.
The method validMoves uses the method standardMoves found in the checker class. The method standardMoves recognizes what diagonally adjacent squares a checker can move based upon what colour the checker is and whether or not it is a king checker. (White checkers can only move up the board, red can only move down, and king checkers of each colour can move both directions) standardMoves returns an ArrayList of Point objects that the method validMoves uses to find the checkerBoardSquares that correspond to those moves.
`

Requirement 2.4 Save a game to be resumed later.
When the user clicks the button "Save Game" on the menu during a game, the series of conditions in mouseClicked identify this and then the saveGame function is executed.

Requirement 3.1 A two player mode is to be implemented where the user interface allows for the swapping of player turns. This has been made possible via the switchTurns method in the Board class, which visually depicts the change of turn, along with a Boolean toggle of a player turn variable to be worked with among the mechanics of the game class.

Requrement 3.2 Single player mode is to be implemented where an AI plays against the user. This has been implemented via an AI class which takes control over the red checker turn upon player turn toggle. This class implements the methods `findWeightOfAllCapturePaths`, `findWeightOfCapturePath` and `findSafetyOnSquare` in order to prioritize which movements are of greater concern.

Evaluation of the Design

This design so far has a fairly good separation of concerns and exhibits a hierarchy.
This design, however, does not limit coupling as much as possible. The Checker class, for example, uses an instance of CheckerBoardSquare class unnecessarily. The Checker class currently accesses the CheckerBoardSquare class to update the images of a checker if it moves or becomes a king. However, this could all be taken out of the Checker class by simply having the place field variable in the Checker Class switch to a Point object that corresponds to its location on the board. The methods in the Checker class responsible for updating images on the CheckerBoardSquare could be entirely removed an executed in the Game class.

Revision History

1. Board Class altered to have new buttons
2. Listener decipherPoint altered to accommodate new buttons
3. Listener has new boolean values singlePlayerMode and player1Turn
4.
    - mouseClicked altered to account for turns
    - Turn doesn’t switch if last move was a capturing move
5. validCheckers
6. validMoves altered to return an array rather than update moves
7. State information has been transferred to a new Class, the Game Class. This is so that the AI and the Listener can access all stat information accordingly, preserving the hierarchy. (The AI would've had to access the listener class otherwise)
8. Addition of the AI class
9. The board class has a method switchTurns that lets the user know whose turn it is.

Testing

In the testing phase, all modifications made to the new design have been represented by bolded lettering indicating either a new input or input response.

Tasks	Output			
	Desired Response	Problems	Unexpected Responses	Pass/Fail
Custom Game	User can place the checkers pieces, one by one for both colours. User is warned when trying to place more than allow number of each colour checkers or if they are placed in an incorrect place.	Users can place checkers where they please; they do not get warning either. Game can be played with this illegal placement.		Fail
Save Game	The current game being played can be saved by the user.	None.		Pass
Playing saved game	Saved game is displayed on board and correct user can make the next move as indicated by the message board.	The message board always indicated that it is 'White's Turn' whether it is or not, however the correct user can make the more. So if it was red's turn, even if they message says white, only the red will be allow to make the next		Fail

		move.		
Multiple saved games	The user is able to save more the one game without the older game is overwritten.	Only one game can be saved, older game is overwritten.		Fail
Two player game	Users take turns making moves, showing them where each checker, when selected, can move. If a user has a move that will take a piece from the opposing team, they game will only allow the user to make that move. When reach the opposing side, the pieces become king pieces and are able to move backwards and forwards.	None	Requires multiple clicks before action is registered.	Fail
One player game	User plays opposing the computer. If the user has a move that will take a piece from the opposing team, they game will only allow the user to make that move. When reach the opposing side, the pieces become king	None	Requires multiple clicks before action is registered.	Fail

	pieces and are able to move backwards and forwards.			
Indicates if a game has been won or lost	When the game comes to an end, the user is notified of it win or loss	User is not notified		Fail
Reset game	All pieces are removed from the board.	None.		

Expected Mouse Input	**Output**			
	Desired Response	**Undesired Response**	**Acceptable response**	**Unknown Response**
In all 'Start Game' states (custom and regular) Menu button 'Reset'	Removes all checker pieces from the board	**An error exception is thrown unless at least one piece is moved or the game is saved**		
Menu button 'Start Game'	In initial state, sets up a standard game with 12 of each colour in their respective places. **In the 'Custom Game' state, initiates the game-play state when there are one or more of each colour of piece on the board.** **In all states the menu display's the 'Reset' and 'Save Game' buttons in place of all other menu buttons**			
Menu button 'Custom Game'	Displays the two buttons for selecting which game piece to place			
Menu button 'For White Checker'	Allows placement of white checkers on checker board and displays a white checker indication message	**Often requires multiple clicks before action is registered**		
Menu Button 'For Red Checker'	Allows placement of red checkers on checker board and displays a red checker indication message	**Often requires multiple clicks before action is registered**		
Black checker	Places a checker in square when in white or red	**Often requires multiple clicks**	In all other states, displays an	

square	checker placement state within the 'Custom Game' state	**before action is registered**	invalid space error in the message bar	
White checker square	Displays an invalid space error in the message bar when in checker placement state (white or red)	**Often requires multiple clicks before action is registered**	Displays an invalid space error in all other states	
Secondary Menu Button 'Save Game'	**Display's a message indicating a current game being stored**			
Moving pieces after entering the 'Start Game' state	**Pieces that are selected have all available moves highlighted. If a selected piece has no available moves, a message is displayed notifying the user that no moves are available. If a piece is jumped, it is removed from the board. If a checker makes its way to its opposite end of the board, it is 'kinged' and is then allowed to move in any which direction.**	**Often requires multiple clicks before action is registered.** **If a checker is available to be taken, but is not, the checker enters a "stuck" state where its "available" moves are still displayed and the checker is allowed to make illegal moves until this erroneous state is exited.**	**If a checker is "stuck" in a movement state, spamming other checkers will eventually allow that "spammed" checker to be moved, thus exiting the locked movement state of the previous checker.**	