Knight's Tour Design Document

By: Aidan Robbins

Firstly this program utilizes awt, events, swing, arraylists, and random.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.ArrayList;
import java.util.Random;
```

The first class is the class Tour which extends JPanel. It has a label to act as the counter and a button for restarting. The main method creates a JFrame for the game and sets close operations, content pane, and packs so that the elements will be properly sized.

```
public class Tour extends JPanel{
private JLabel counter;
private JButton restart;

    public static void main(String[] args) {
        JFrame win = new JFrame ("Knight's Tour");
        Tour tour = new Tour();
        win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        win.setVisible(true);
        win.setContentPane(tour);
        win.pack();
    }
```

The Constructor sets the layout as null because I manually set them. It then creates a new instance of the Game class and adds the game, counter, and restart button and then sets their dimensions.

```
    public Tour() {
        setLayout(null);
        setPreferredSize(new Dimension(600, 660));
        Game game = new Game();
        add(game);
        add(counter);
        add(restart);
        game.setBounds(0, 60, 600, 600);
        counter.setBounds(0, 30, 600, 30);
        restart.setBounds(0,0, 600, 30);
    }
```

Before getting to the two major classes there is a small class called Move which keeps track of a from row, from column, to row, and to column. It includes a constructor and getters. This class is used to make arrays and arraylists of moves to keep track of legal moves.

```
public class Move{
    int fromR;
    int fromC;
    int toR;
    int toC;

    Move(int fromR, int fromC, int toR, int toC){
        this.fromR = fromR;
        this.fromC = fromC;
        this.toR = toR;
        this.toC = toC;
    }

    public int getFromR() {
        return fromR;
    }

    public int getFromC() {
        return fromC;
    }

    public int getToR() {
        return toR;
    }

    public int getToC() {
        return toC;
    }


}
```

Next is the class Tiles. Firstly I set three final ints to represent the possible states a given tile can be in with 0 representing an open tile, 1 representing an occupied tile ie where the Knight is, and 2 representing tiles that have already been visited. I also create a 2D array to represent the board. I also create a random r.

```
public class Tiles{
    final int open = 0;
    final int occupied = 1;
    final int visited = 2;
    int[][] board;
    Random r = new Random();
```

The setUp() method initializes the board. It iterates through all rows and columns setting each to 0. It then uses the random number generator to select a random tile which it sets to 1 to represent where the knight begins. The constructor initializes the board array and calls setUp(). There is also a tileStatus method which returns the int at some tile.

```
void setUp() {
    for(int row = 0; row<8; row++) {
        for(int col = 0; col <8; col++) {
            board[row][col] = 0;
        }
    }
    board[r.nextInt(8)][r.nextInt(8)] = occupied;
}//end set up

Tiles(){
    board = new int[8][8];
    setUp();
}

int tileStatus(int r, int c) {
    return board[r][c];
}
```

0	0	0	0	0	0	0	0
0	2	0	0	0	0	0	0
0	0	0	2	0		0	0
2	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Visualization of a possible board array

The Move method takes a Move object as input and sets the getTos as occupied and the forms as visited. It is only ever called if the canMove method returns true. The canMove takes in the coordinates of a move and makes sure that the row and column it plans to move to are on the board.

```
public void Move(Move move) {

    board[move.getToR()][move.getToC()] = occupied;
    board[move.getFromR()][move.getFromC()] = visited;

}

boolean canMove(int r1, int c1, int r2, int c2) {
    if(r2 >= 8 || r2 <0 || c2 >= 8 || c2 <0) {
        return false; //off board
    }
    else {
        return true; //legal
    }
}
```

A major method is the getLegals() method which will return an array of Move objects. Firstly in itializes the kRow and kCol as -1. These are to represent where the knight is. It then creates an ArrayList of Moves. Next it iterates through the board to find the knight and sets kRow and kCol accordingly. Next it checks all 8 possible moves a knight can make using the canMove method and if it returns true that move is added to the ArrayList. After this has been checked if there are for some reason no moves it returns null otherwise it converts the ArrayList to an Array by making an array of the same length. It then moves all the moves into that array and return it.

```
Move[] getLegals() {
    int kRow = -1;
    int kCol = -1;
    ArrayList<Move> moves = new ArrayList<Move>();
    for(int row =0; row<8; row++) {
        for(int col =0; col<8; col++) {
            if(board[row][col] == occupied) {
                kRow = row;
                kCol = col;
            }
        }
    }

    //checking all of the possible moves a knight can make
            if(canMove(kRow, kCol, kRow + 2, kCol +1)) {
                moves.add(new Move(kRow, kCol, kRow +2, kCol +1));
            }
            if(canMove(kRow, kCol, kRow + 2, kCol -1)) {
                moves.add(new Move(kRow, kCol, kRow +2, kCol -1));
            }
            if(canMove(kRow, kCol, kRow - 2, kCol +1)) {
                moves.add(new Move(kRow, kCol, kRow -2, kCol +1));
            }
            if(canMove(kRow, kCol, kRow - 2, kCol -1)) {
                moves.add(new Move(kRow, kCol, kRow -2, kCol -1));
            }
            if(canMove(kRow, kCol, kRow + 1, kCol +2)) {
                moves.add(new Move(kRow, kCol, kRow +1, kCol +2));
            }
            if(canMove(kRow, kCol, kRow - 1, kCol +2)) {
                moves.add(new Move(kRow, kCol, kRow -1, kCol +2));
            }
            if(canMove(kRow, kCol, kRow + 1, kCol -2)) {
                moves.add(new Move(kRow, kCol, kRow +1, kCol -2));
            }
            if(canMove(kRow, kCol, kRow - 1, kCol -2)) {
                moves.add(new Move(kRow, kCol, kRow -1, kCol -2));
            }

    if(moves.size() ==0) {
        return null;
    }
    else {
        Move[] movesArray = new Move[moves.size()];
        for(int i = 0; i <moves.size(); i++) {
            movesArray[i] = moves.get(i);
        }
        return movesArray;
        }
    //returns an array of all legal moves

}//end get legals
```

This is the final method of the Tiles class so we will move onto the Game class which extends JPanel and implements both the MouseListener and ActionListener. This class has a Tiles object called game, a boolean to keep track of if the game is in progress, a selected row int and selected column int, an array of Move objects called legals which tracks legal moves, an int called visits to keep track of moves made and an ArrayList of points for drawing progress.

```
private class Game extends JPanel implements MouseListener, ActionListener{
    Tiles game;
    boolean inProgress;
    Move[] legals;
    int visits = 1;
    ArrayList<Point> points = new ArrayList<Point>();
```

The constructor adds a mouse listener, creates the counter label and restart button. It adds the button's listener and creates a new Tiles() before calling the method doNew().

```
    public Game() {
        setBackground(Color.BLACK);
        addMouseListener(this);
        counter = new JLabel("", JLabel.CENTER);
        restart = new JButton("Restart");
        restart.addActionListener(this);
        game = new Tiles();
        doNew();
    }
```

The action performed method is for the restart button and when the button is pressed clears the points ArrayList if it has any points in it, sets the visits back to 1, sets inProgress to false and then calls the doNew() method in order to start over. The doNew() method checks if a game is in progress and if it isn't class game.setUp() to initialize the board. It also calls game.getLegals to get the first set of legal moves. It then sets the counter text to "Tiles Visited: " + visits which at this point is 1 for the first tile. Finally it sets inProgress to true and repaints.

```
    public void actionPerformed(ActionEvent e) {
        switch (e.getActionCommand()) {
        case "Restart":
            if(points.size() > 0) {
                points.clear();
            }
            visits = 1;
            inProgress = false;
            doNew();
            // button which restarts the game, sets visits back to one and clears the point arraylist
        }
    }


    public void doNew() {
        if(inProgress != true) {
            game.setUp();
            legals = game.getLegals();
            counter.setText("Tiles Visited: " + visits);
            inProgress = true;
            repaint();
        }
    } // end do new
```

The endGame() method sets the counter to have a congratulations method which tells you how many moves you made and sets the progress to false. This gets called if a check is passed in another method which will then call endGame(). The method Click takes a point and gets the legal moves. It then iterates through the legal move array to see if the clicked point matches any legal move, if so it calls the doMove method in order to perform that move.

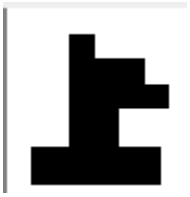
```
public void endGame() {
    counter.setText("Congratulations! You have completed the tour in " + visits + " moves.");
    inProgress= false;
}

public void click(int r, int c) {
    legals = game.getLegals();
    for(int i =0; i < legals.length; i++) {
        if(legals[i].getToR() == r && legals[i].getToC() == c) {
            doMove(legals[i]);
        }
    }
}
```

doMove takes a Move object and utilizes the move method from the Tiles class, it then updates the legals array and increases the visits counter by 1 and updates the counter text. It then calls the hasWon() method to check if the user has completed the tour. Finally it adds the point the move started at to the points ArrayList so it can be drawn and then calls repaint. The hasWon() method checks if the game is in progress and if so iterates through the board checking the status of each via the tileStatus method from the Tiles class. If it finds any tiles equal to game.open (0 as set in Tiles) it returns to tell the rest of the code that there is at least one tile left to visit. If it reaches the end and doesn't find any open tiles it sets inProgress to false and calls endGame().

```
    public void doMove(Move move) {
        game.Move(move);
        legals = game.getLegals();
        visits = visits + 1;
        counter.setText("Tiles Visited: " + visits);
        hasWon();
        points.add(new Point(move.getFromC() , move.getFromR() ));
        repaint();
    }

    public void hasWon() {
        if(inProgress) {
        for(int row =0; row <8; row++) {
            for(int col =0; col <8; col++) {
                if(game.tileStatus(row, col) == game.open) {
                    return;
                }
            }
            }
        }
        inProgress = false;
        endGame();
    }
```

The paintComponent first sets the occupied col and row ints as -1. Then it draws a checkerboard of gray and white tiles. It sees if col%2 and row%2 are equal and if so that tile will be gray and if not it will be white. Then it checks if that tile is occupied, if so OccRow and OccCol will be set accordingly for drawing lines later in the component. It thend draws a little black icon of a knight piece at that tile.

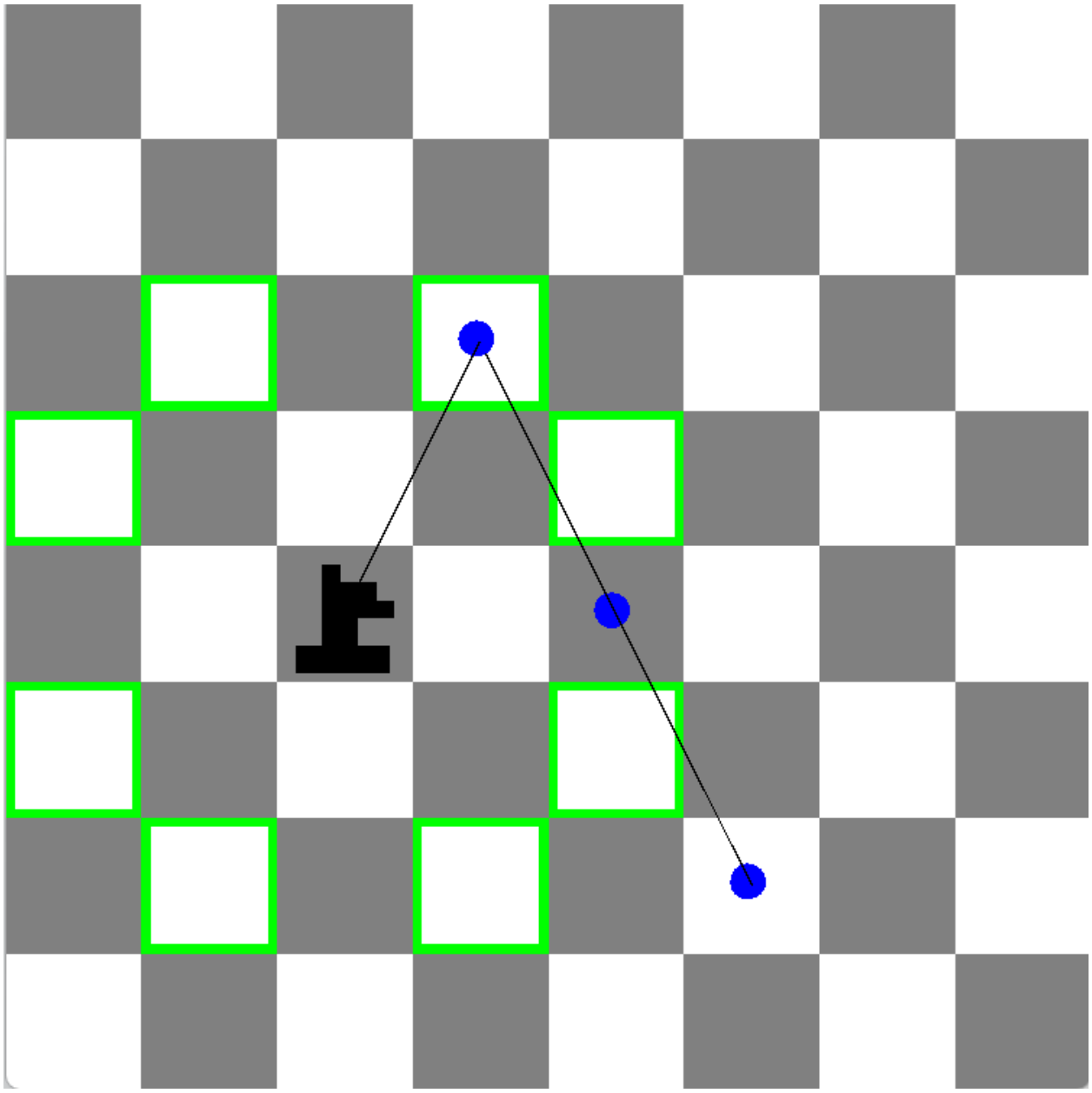
```
occRow = row;
g.setColor(Color.BLACK);
g.fillRect( col *75 +10 , row *75 + 55, 52, 15);
g.fillRect( col *75 +25 , row *75 + 20, 20, 50);
g.fillRect( col *75 +45 , row *75 + 20, 10, 10);
g.fillRect( col *75 +45 , row *75 + 30, 20, 10);
g.fillRect( col *75 +25 , row *75 + 10, 10, 10);
```



Next it checks if the game is in progress and if so iterates through the legals array and draws a green rectangle with a smaller rectangle inside of it that is either gray or white determined by the same method as before. This creates a highlighted rectangle to display legal moves.
Finally it checks that the points ArrayList has data in it and then goes through that list and draws connecting lines between all points. It does this by first checking if k-1 (k being the integer increasing through the loop) is larger than 0 and if so draws from k's coordinates to k-1's coordinates. The data in these are in rows and columns so to be drawn they are multiplied by 75 and 37 is added to make them in the middle of their tile. If the size of points is 2 it draws from points.get(0) to points,get(1). This solved a bug I had where on the second click specifically the first line would disappear. Finally it draws blue circles in the center of each tile in points to convey where the knight has been and it then draws a line from the last point to where the knight is currently.

```
public void paintComponent(Graphics g) {
    int OccCol = -1;
    int OccRow = -1;
    //first draws a checkerboard
    for(int row =0; row <8; row ++) {
        for(int col =0; col <8; col++) {
            if(row%2 == col%2) {
                g.setColor(Color.GRAY);
            }
            else {
                g.setColor(Color.WHITE);
            }
            g.fillRect(col * 75, row * 75, 75, 75);

            //draws the knight and saves the coordinates
            if(game.tileStatus(row, col) == game.occupied) {
                OccCol = col;
                OccRow = row;
                g.setColor(Color.BLACK);
                g.fillRect( col *75 +10 , row *75 + 55, 52, 15);
                g.fillRect( col *75 +25 , row *75 + 20, 20, 50);
                g.fillRect( col *75 +45 , row *75 + 20, 10, 10);
                g.fillRect( col *75 +45 , row *75 + 30, 20, 10);
                g.fillRect( col *75 +25 , row *75 + 10, 10, 10);
            }
        }
    }

    if (inProgress) {
        //highlighting the legal moves
        for(int i = 0; i < legals.length; i++) {
            g.setColor(Color.GREEN);
            g.fillRect(legals[i].getToC() *75 , legals[i].getToR() * 75, 75, 75);
            if(legals[i].getToR()%2 == legals[i].getToC()%2) {
                g.setColor(Color.GRAY);
                g.fillRect(legals[i].getToC() *75 +5  , legals[i].getToR() * 75 + 5, 65, 65);

            }
            else {
                g.setColor(Color.WHITE);
                g.fillRect(legals[i].getToC() *75 +5  , legals[i].getToR() * 75 + 5, 65, 65);
            }

            }

        }
```

```
if( points != null) {
    //drawing the lines to track the path also draws blue dots to denote previously visited tiles. Then draws a line to the current position and redraws the knight in case
    //it is at a previously visited tile otherwise it would be covered by a circle
    g.setColor(Color.BLACK);

    for(int k =0; k < points.size(); k++) {
        if(k - 1 > 0) {
            g.drawLine((points.get(k).getX() * 75) +37 , (points.get(k).getY() * 75) + 37, (points.get(k -1).getX() * 75) +37 , (points.get(k -1).getY() * 75) +37 );
            g.drawLine(points.get(0).getX() *75 + 37, points.get(0).getY() * 75 +37, points.get(1).getX() *75 + 37, points.get(1).getY() * 75 +37);
        }
        if(points.size() == 2) {
            g.drawLine(points.get(0).getX() *75 + 37, points.get(0).getY() * 75 +37, points.get(1).getX() *75 + 37, points.get(1).getY() * 75 +37);
        }
        g.setColor(Color.BLUE);
        g.fillOval((points.get(k).getX() * 75) +25 , (points.get(k).getY() * 75) + 25, 20, 20);
        g.setColor(Color.BLACK);
        g.drawLine(points.get(points.size() -1).getX() * 75 + 37, points.get(points.size() -1).getY() * 75 + 37, OccCol * 75 + 37, OccRow * 75 + 37);
        g.setColor(Color.BLACK);
        g.fillRect( OccCol *75 +10 , OccRow *75 + 55, 52, 15);
        g.fillRect( OccCol *75 +25 , OccRow *75 + 20, 20, 50);
        g.fillRect( OccCol *75 +45 , OccRow *75 + 20, 10, 10);
        g.fillRect( OccCol *75 +45 , OccRow *75 + 30, 20, 10);
        g.fillRect( OccCol *75 +25 , OccRow *75 + 10, 10, 10);
    }
}


}// end paint
```



(All of the paint components together.)

Lastly in the game class is the mousePressed method which converts a click to a row and column, then if it is within the board it calls the click method on the row and column of the mouse click.

```
public void mousePressed(MouseEvent e) {
    if(inProgress) {
        int col = e.getX() / 75;
        int row = e.getY() / 75;
        if(col >= 0 && col < 8 && row >= 0 && row < 8) {
            click(row, col);
        }
    }
}

public void mouseReleased(MouseEvent e) {
}
public void mouseClicked(MouseEvent e) {
}
public void mouseEntered(MouseEvent e) {
}
public void mouseExited(MouseEvent e) {
}
```

Finally there is a small Point class which just keeps track of an x and y which is used for the points ArrayList in the game class.

```
public class Point{
    int x;
    int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }


}
```

All taken together and running it creates a game which looks like the following: