Java Source Code

Main:

```
package assignment1;

import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;

/**
 * <h2>Main Class</h2>
 * <p>
 * Main class where the program begins.
 * The class sets up what the look and feel for the JFrame
 * should be for the program and opens it in a separate thread.
 * </p>
 * <p>
 * My implementation of the SwingUtilities and use of
 * the LookAndFeel may not be exactly correct. Due to the
 * time scale of this project, I decided to re-use old
 * code from my past project 'Blockmation', from my first
 * year in University. This is an artifact of that code that
 * I remember working.
 * <br />
 * Should there have been more time to work on this assignment,
 * I would have fully investigated these and also worked on making
 * the GUI more user friendly, informative as to what Cell is
 * which and what Cells had just been updated and how exactly.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class Main {
    @SuppressWarnings("unused")
    private static Driver driver;
    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (ClassNotFoundException e) {
            JOptionPane.showMessageDialog(new JFrame(), "Class not found!",
                    "Error: Look and Feel - Class Not Found",
                    JOptionPane.ERROR_MESSAGE);
        } catch (InstantiationException e) {
            JOptionPane.showMessageDialog(new JFrame(),
                    "Not able to Instantiate Look and Feel!",
                    "Error: Look and Feel - Unable to Instantiate",
                    JOptionPane.ERROR_MESSAGE);
        } catch (IllegalAccessException e) {
            JOptionPane.showMessageDialog(new JFrame(),
                    "Illegal Access Exception!",
                    "Error: Look and Feel - Illegal Access Exception",
                    JOptionPane.ERROR_MESSAGE);
        } catch (UnsupportedLookAndFeelException e) {
            JOptionPane.showMessageDialog(new JFrame(), "Look and Feel "
                    + UIManager.getCrossPlatformLookAndFeelClassName()
                    + " not supported!",
                    "Error: Look and Feel - Unsupported Look and Feel",
                    JOptionPane.ERROR_MESSAGE);
        }
        makeApp();
    }

    /**
     * <p>
     * Creates the application with invokeLater().
     * </p>
     */
    public static void makeApp() {
        try {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    build();
                }
            });
        } catch (Exception e) {
            JOptionPane.showMessageDialog(new JFrame(),
                    "Exception encountered on attempt to build application",
                    "Error - Not able to build application",
                    JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * <p>
     * Makes a new instance of <code>SolverFrame</code> that opens the Sudoku Solver
     * window.
     * </p>
     */
    public static void build() {
        driver = new Driver();
    }
}
```

Driver:

```
package assignment1;

/**
 * <h2>Driver</h2>
 * <p>
 * A 'between' class that goes between the main classes the
 * Main method class, keeping the classes in use private.
 * <br />
 * A reference to SudokuSolver is passed to the window
 * to begin with and help setup the SolverFrame GUI.
 * This will be overwritten later as a
 * new Sudoku puzzle is opened however.
 * </p>
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class Driver {
    @SuppressWarnings("unused")
    private SolverFrame window;
    private SudokuSolver solver;

    /**
     * <p>
     * Creates instances of the solver and GUI viewer.
```

```
         * </p>
         */
        public Driver(){
            solver = new SudokuSolver();
            window = new SolverFrame(solver);
        }
    }
```

SudokuSolver:

```
package assignment1;

/**
 * <h2>SudokuSolver</h2>
 * <p>
 * The top level of the model, from where the Model is created and it's
 * attributes for the particular Sudoku puzzle loaded are applied, along with
 * any actions to take during solving. <br />
 * This class holds the solving method that calls on each individual solving
 * algorithm as and when they are needed from the SudokuModel.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class SudokuSolver {

    private SolverModel grid;
    private Result result;
    private static final int SUDOKU_SPACES = 9;
    /**
     * mRows and mColumns are the 'modifiers' used
     * in Pointing Pairs, to determine if the method
     * is being used on Rows or Columns.
     * More can be seen on this in SudokuModel.
     */
    private static final int mRows = 1;
    private static final int mColumns = 3;
    private int steps;
    private StringBuffer sb;

    /**
     * <p>
     * Constructs the SolverModel and sets it up to have a set of blank Cells in
     * preparation for loading .sud files with come cells filled.
     * </p>
     */
    SudokuSolver() {
        grid = new SolverModel();
        sb = new StringBuffer();
        result = new Result();
        grid.setCells(new int[9][9]);
        steps = 0;
    }

    /**
     * <p>
     * Sends a 2D Character array to a method to be converted into a 2D Integer
     * array and then sends this as an argument to the SudokuModel class to be
     * set as the current Cells to work with for solving a puzzle. <br />
     * The char array comes from reading in a File where all text is stored as
     * char/Strings and needs converting in order to be used in the sudoku grid
     * as Integers with values.
     * </p>
     *
     * @param lines
     *            - 2D Character array that will be converted into an array of
     *            Integers in this class (see 'convertChar(char[][])') and sent
     *            to the SudokuModel.
     */
    public void setGrid(char[][] lines) {
        grid.setCells(this.convertChar(lines));
    }

    /**
     * <p>
     * A request to get the SudokuModel's grid, used by classes holding
     * reference to this class but not directly to the SudokuModel.
     * </p>
     *
     * @return The Grid from SudokuModel
     */
    public Cell[][] getGrid() {
        return grid.getGrid();
    }

    /**
     * <p>
     * When the request to 'take a step' in solving the Sudoku puzzle is called,
     * this method runs through the order of priority the solving algorithms
     * should be called in, and is written such that only one step can be taken
     * at a time, and only after those that have come before it have failed to
     * succeed.
     * </p>
     * <p>
     * In light of Complexity, I feel there is likely a much more efficient way
     * of handling this request than 'if this, then do this', as if we wish to
     * use the most advanced techniques, we must first attempt every other
     * technique before it, adding complexity to each failed result. This would
     * be addressed with more time on the project, however, my main goal was to
     * address the data structures and individual solving algorithms on their
     * own.
     * </p>
     * <p>
     * Each method creates it's own Result from running the method. More
     * information can be seen about this in the 'Result' class. However, these
     * contain information on whether the operation was a success, what message
     * to show the user and any Cells that were affected with updated
     * Values/solved. <br />
     * With more time, I would expand into highlighting in text where the Cells
     * were updated, and also visually show which candidates would be removed.
     * </p>
     *
     * @return A String containing the step taken in order to solve the next
     *         section of the grid.
     */
    public String takeStep() {
        result = grid.removeCollectionCandidates();
        if (!result.getSuccess()) {
            result = grid.hiddenSingles();
            if (!result.getSuccess()) {
                result = grid.nakedSingles();
                if (!result.getSuccess()) {
                    result = grid.pointingPairs(mRows);
                    if (!result.getSuccess()) {
                        result = grid.pointingPairs(mColumns);
                        if (!result.getSuccess()) {
                            result = grid.nakedPairsAndTriples();
                            if (!result.getSuccess()) {
                                result = new Result();
                            }
                        }
                    }
                }
            }
        }
        successfulMethod(result.getMessage());
        return sb.toString();
    }

    /**
     * <p>
     * Defines what a 'row', 'column' and 'block' is, adding each of them to a
     * single LinkedList holding all types. <br />
     * Ordering: Row, Column, Row, Column, etc, ending with Column, Block,
     * Block.. Block.
     * </p>
     */
    public void setup() {
        grid.defineRowsAndColumns();
        grid.defineBlocks();
    }

    /**
     * <p>
     * After being passed a 2D array of Characters, the method creates a new 2D
     * array of Integers and begins converting the values. <br />
     * The conversion is as simple as getting the numerical value of the
     * character, rather than it's Hex, Decimal or Octal value. The Numeric
     * value is quite literally 1 = 1, 2 = 2, 3 = 3, etc. <br />
     * Any blank spaces are converted to -1, which are dealt with when passed to
     * the SolverModel.
     * </p>
     * <p>
     * The filled and resulting 2D array of Integers converted from the 2D
     * Character array is returned to where it was called.
     * </p>
     *
     * @param lines
     * @return
     */
    public int[][] convertChar(char[][] lines) {
        int[][] numbers = new int[SUDOKU_SPACES][SUDOKU_SPACES];
        for (int i = 0; i < SUDOKU_SPACES; i++) {
            for (int j = 0; j < SUDOKU_SPACES; j++) {
                numbers[i][j] = Character.getNumericValue(lines[i][j]);
            }
        }
        return numbers;
    }

    /**
     * <p>
     * Asks the SudokuModel if the puzzle is solved and returns a boolean answer
     * based on the response received.
     * </p>
```

```
     *
     * @return
     */
    public boolean isSolved() {
        if (grid.solved()) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * @return the amount of steps taken to completion so far.
     */
    public int getSteps() {
        return steps;
    }

    /**
     * <p>
     * Builds and returns the message based on the successful solving steps
     * taken.
     * </p>
     *
     * @param msg
     */
    public void successfulMethod(String msg) {
        steps++;
        sb.append("Step: ");
        sb.append(steps);
        sb.append(msg);
    }

    /**
     * @return - The Result currently stored here from the last successful
     *         algorithm operation.
     */
    public Result returnResult() {
        return result;
    }
}
```

SudokuSolver:

```
package assignment1;

import java.util.LinkedList;
import java.util.Stack;

/**
 * <h2>SolverModel</h2>
 * <p>
 * Deals with all operations upon the values and candidates of the Cells, in
 * respect to their location and status within the 'Sudoku' grid. <br />
 * Many different algorithms are used in order to solve a variety of different
 * sudoku puzzles, some easy and some a little tougher. This is the 'heart' of
 * the SudokuSolver.
 * </p>
 * <p>
 * Many algorithms are self contained, and attempt to use the same code for both
 * Rows and Columns. Some methods grew larger and needed to be split into
 * multiple methods (i.e. PointingPairs - Large If statement conditions), for
 * ease of maintainability and explanation.
 * </p>
 * <p>
 * Collection tends to refer to a Row, Column or Block. <br />
 * Each method returns a 'Result', containing it's success, any cells affected
 * and any message attached to the success. More can be seen on this in the
 * JavaDoc for 'Result.class'.
 * </p>
 * 
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class SolverModel {

    private Cell[][] grid = new Cell[9][9];
    private Cell cell;
    private Result result;
    private static final int SUDOKU_SPACES = 9;
    private static final int BLOCK_SPACES = 3;
    private static final int MAX_CELLS = 81;
    private LinkedList<Cell> collection;
    private LinkedList<LinkedList<Cell>> allCollections;
    private StringBuffer sb;

    /**
     * <p>
     * Constructor prepares class for use by making new instances of
     * 'allCollections', which holds all rows, columns and blocks as ordered
     * lists.
     * </p>
     */
    public SolverModel() {
        allCollections = new LinkedList<LinkedList<Cell>>();
    }

    /**
     * <p>
     * What values, including 0, to set to particular Cells to build the current
     * Sudoku puzzle.
     * </p>
     * 
     * @param lines
     *            - Cells as read in by FileHandler.
     */
    public void setCells(int[][] lines) {
        for (int i = 0; i < SUDOKU_SPACES; i++) {
            for (int j = 0; j < SUDOKU_SPACES; j++) {
                cell = new Cell(lines[i][j], i, j);
                grid[i][j] = cell;
            }
        }
    }

    /**
     * <p>
     * Through the use of a nested for loop, gets what would be each Cell in a
     * 'Row' or 'Column' (depdning on 'direction' of i and j) and adds them into
     * the overall list of collections (Row, Column, Block).
     * </p>
     */
    public void defineRowsAndColumns() {
        for (int i = 0; i < SUDOKU_SPACES; i++) {
            LinkedList<Cell> collectionRow = new LinkedList<Cell>();
            LinkedList<Cell> collectionColumn = new LinkedList<Cell>();
            for (int j = 0; j < SUDOKU_SPACES; j++) {
                collectionRow.add(grid[i][j]);
                collectionColumn.add(grid[j][i]);
            }
            allCollections.add(collectionRow);
            allCollections.add(collectionColumn);
        }
    }

    /**
     * <p>
     * Similar to defineRowsAndColumns, yet involving slightly different use of
     * nested for loops, as each block starts, continues for 3 cells, then moves
     * down a row (or across a column, depending how you build it. This example
     * starts and iterates over rows).
     * </p>
     * <p>
     * By using multiple nested for loops, and knowing that each block is size
     * of 3, we can increment through the Cells relatively easy. <br />
     * For example, we know that Block 2 (last on top Row), starts at, and
     * includes, cell (0, 6), which would be reached by rowStart = n *
     * Block_Spaces, or, 2 * Block_Spaces = 6. From there, it's a simple case of
     * iterating through the grid to grab the rows in each block below the
     * starting block when we know where to start and that no block goes more
     * than +3 in either rows or columns from the start.
     * </p>
     */
    public void defineBlocks() {
        int rowStart;
        int columnStart;
        int block = 0;
        for (int n = 0; n < BLOCK_SPACES; n++) {
            for (int k = 0; k < BLOCK_SPACES; k++) {
                rowStart = n * BLOCK_SPACES;
                columnStart = k * BLOCK_SPACES;
                collection = new LinkedList<Cell>();
                for (int i = rowStart; i < (rowStart + BLOCK_SPACES); i++) {
                    for (int j = columnStart; j < (columnStart + BLOCK_SPACES); j++) {
                        collection.add(grid[i][j]);
                        grid[i][j].setBlock(block);
                    }
                }
                allCollections.add(collection);
                block++;
            }
        }
    }

    /**
     * @return The full list of Row, Column and Blocks.
     */
    public LinkedList<LinkedList<Cell>> getAllCollections() {
        return allCollections;
    }

    /**
     * <p>
     * Removes candidates from each Cell based on what candidates currently
     * exist in the grid. <br />
     * Searches the grid and adds existing values to a list, then goes through
     * this list and removes any cells that hold them as candidates in each type
     * of collection.
     * </p>
     *
     * @return Result of operation.
     */
    public Result removeCollectionCandidates() {
        result = new Result();
        for (LinkedList<Cell> currentCollection : allCollections) {
            LinkedList<Integer> existsInCollection = new LinkedList<Integer>();
            for (Cell currentCell : currentCollection) {
                if (currentCell.hasValue()) {
                    existsInCollection.add(currentCell.getValue());
                }
            }
            for (Cell currentCell : currentCollection) {
                for (int candidate : existsInCollection) {
                    if (!currentCell.hasValue()) {
                        if (currentCell.hasCandidate(candidate)) {
                            currentCell.removeCandidate(candidate);
                            result.setSuccess(true);
                            result.setMessage("<br />Checked existing values<br
/>Removed candidates<br />");
                        }
                    }
                }
            }
        }
        return result;
    }

    /**
     * <p>
     * Standard Sudoku technique involving looking through each Cell in the grid
     * in a particular collection and finding those with a value that appears
     * within that Cell and only that Cell in that one particular collection,
     * even if it belongs to another collection that has many cells looking for
     * this candidate. <br />
     * Due to only one cell available for this candidate, it is only possible
     * for this cell to be this value, and so it is set to the cell.
     * </p>
     *
     * @return Result of operation.
     */
    public Result nakedSingles() {
        result = new Result();
        Stack<Cell> stack = new Stack<Cell>();
        for (LinkedList<Cell> currentCollection : allCollections) {
            /*
             * For all collections, and all cells in that collection, iterate
             * and find the ones who have the candidate of the current valueNum
             * we are looking for.
             */
            for (int valueNum = 1; valueNum <= SUDOKU_SPACES; valueNum++) {
                stack.clear(); // Being sure to keep a clean stack on the start
                               // of new candidate/value checks.
                for (Cell currentCell : currentCollection) {
                    if (currentCell.hasCandidate(valueNum)) {
                        /*
                         * If a Cell is found to have the current searched for
                         * value, add Cell to stack.
```

```java
                         */
                        stack.push(currentCell);
                    }
                }
                if (stack.size() == 1) {
                    /*
                     * If the stack size is one, we know that it must contain
                     * only one Cell from that collection currently searched,
                     * and so that Cell must be the only cell that can hold this
                     * particular value, so set it.
                     */
                    Cell c = stack.pop();
                    c.setValue(valueNum);
                    // Result generation.
                    result.setSuccess(true);
                    sb = new StringBuffer();
                    sb.append("<br />Checked for Naked Singles");
                    appendCellInfo(c);
                    result.setMessage(sb.toString());
                    result.addAffected(c);
                    return result;
                }
            }
        }
        return result;
    }

    /**
     * <p>
     * The technique of nakedPairsAndTriples relies upon finding Cells who have
     * only two or three values, shared between one another, within a
     * collection. <br />
     * We can be assured that if we find two cells in the collection who share
     * the two same candidates, then the values must belong to these cells and
     * no others, so those candidates are removed from other cells in the
     * collection. <br />
     * In the case of finding three cells with three values, the same applies.
     * However, the same is also true if we find a cell that may have three
     * candidates, shared between two cells with only two candidates (e.g. {3,
     * 6, 9}, {6,9}, {3,9}). Even with this in case, we know that out of all
     * cells, only these three can really contain these values, as other cells
     * have other options. Once again, this leads to removing, this time three,
     * candidates from the other cells in the collection. <br />
     * The other possibility is that three cells appear with three candidates
     * split between them, but no one cell has three candidates to itself. My
     * algorithm here is not strong enough handle these situations. (e.g. {3,4},
     * {3,9}, {4,9} in the same collection).
     * </p>
     *
     * @return Result of operation
     */
    public Result nakedPairsAndTriples() {
        result = new Result();
        for (LinkedList<Cell> currentCollection : allCollections) {
            for (int i = 0; i < SUDOKU_SPACES - 1; i++) {
                cell = currentCollection.get(i);
                if (!cell.hasValue()) {
                    if ((cell.getNumCandidates() > 1)
                            && (cell.getNumCandidates() < 4)) {
                        /*
                         * For each cell in each collection, find unsolved
                         * cells, if they have 2 or three candidates, add their
                         * candidates to a list of candidates to remove, and add
                         * the cell to a list of cells NOT to be altered when
                         * changing collection candidates.
                         */
                        LinkedList<Integer> toRemove = cell.returnCandidates();
                        LinkedList<Cell> doNotTouch = new LinkedList<Cell>();
                        doNotTouch.add(cell);

                        for (Cell otherCell : currentCollection) {
                            /*
                             * For all possible candidates, look through cells
                             * in our collection (that aren't our current first
                             * found cell with 2/3 candidates), and check to see
                             * if they have 2/3 candidates too. If they do,
                             * check if we have 2 or 3 matches to the current
                             * candidates to remove, in order to 'pair' up Cells
                             * with the same candidates.
                             */
                            if (!otherCell.equals(cell)) {
                                if (cell.matchingCandidates(otherCell
                                        .returnCandidates()) > 1
                                        && (cell.matchingCandidates(otherCell
                                                .returnCandidates()) < 4)
                                        && (otherCell.getNumCandidates() < 4)) {
                                    LinkedList<Integer> foundCandidates = otherCell
                                            .returnCandidates();
                                    /*
                                     * If some candidates are different, yet
                                     * there are still matches (e.g. {3,5,7} and
                                     * {3,7}), add the extra candidate to the
                                     * potential list of candidates to be
                                     * removed. Then add the newly found
                                     * potential pair to the list of candidates
                                     * not to be altered.
                                     */
                                    for (int candidate : foundCandidates) {
                                        if (!toRemove.contains(candidate)) {
                                            toRemove.add(candidate);
                                        }
                                    }
                                    doNotTouch.add(otherCell);
                                }
                            }
                        }
                        /*
                         * Check that our list of candidates to remove isn't
                         * empty, then if it is, send values to be evaluated and
                         * have candidates removed in another function for ease
                         * of maintainability. Then generate results.
                         */
                        if (!toRemove.isEmpty()) {
                            if(doNotTouch.size() == toRemove.size()){
                            result.setSuccess(nakedPairsAndTriplesRemoveCandidates(
                                    currentCollection, toRemove, doNotTouch));
                            result.setMessage("<br />Checked for Naked Pairs/Triples<br
/>Removed Candidates<br />");
                        }
                        }
                        /*
                         * Return lists to initial state to not confuse new
                         * pairs and collections with old pairs and collections.
                         */
```

```
                    toRemove = null;
                    doNotTouch.clear();
                }
            }
        }
    }
    return result;
}

/**
 * <p>
 * Part of the nakedPairsAndTriples algorithm, we first go through each
 * candidate for potential removal and with this look at each cell in the
 * collection. <br />
 * For each cell, if it is unsolved, is not one of the Cells not to be
 * altered and that the amount of candidates to remove is equal to the
 * amount of cells not to be altered, remove the candidate from the cell.
 * </p>
 * <p>
 * The check of size of the list of Cells not to be altered versus the
 * amount of candidates to be removed is very important. It is this check
 * that ensures if we are removing anything, it is no more and no less than
 * we have pairs. If it were otherwise, it would mean that the values do not
 * work as pairs/triples.
 * </p>
 *
 * @param currentCollection
 *            - Current Row/Column/Block
 * @param toRemove
 *            - candidates for potential removal
 * @param doNotTouch
 *            - Cells not to be altered
 * @return Result of operation.
 */
public boolean nakedPairsAndTriplesRemoveCandidates(
        LinkedList<Cell> currentCollection, LinkedList<Integer> toRemove,
        LinkedList<Cell> doNotTouch) {
    boolean success = false;
    for (int candidate : toRemove) {
        for (Cell currentCell : currentCollection) {
            if (!currentCell.hasValue()){
                    if(!doNotTouch.contains(currentCell)) {
                currentCell.removeCandidate(candidate);
                success = true;
            }
            }
        }
    }
    return success;
}

/**
 * <p>
 * Checks for pairs within a row or column that 'point'
 * towards other Cells that hold candidates to be removed,
 * as the pair itself hold the candidates to be removed.
 * </p>
 * <p>
 * A 'pair' is looked for along the row or a column or
 * each individual Block. If they are found to have matching
 * candidates, and only those have the candidate in the
 * individual block, we know we can remove the candidates
```

```
     * from all other cells in the row/column, that the cells
     * appear in, outside of the box. They are 'pointing' at
     * cells that need candidates removed, hence the name.
     * </p>
     * <p>
     * The 'mod' parameter is a modifier that should either be
     * a 1 (rows) or 3 (columns). Using simple arithmetic, and
     * knowing that the size of a sudoku grid is always 9x9, a
     * block is always 3x3 and there is also a static amount of
     * Cells, columns, rows and blocks, the behaviour of the
     * method can be altered using this modifier.
     * </p>
     * @param mod - A modifier, either 1 or 3, representing rows
     * or columns, respectively.
     * @return The result of the operation
     */
    public Result pointingPairs(int mod) {
        result = new Result();
        String type = "Columns";
        if (mod == 1) {
            type = "Rows";
        }
        boolean removeCandidates = false;
        int modifier = mod;
        /*
         * For 'all blocks', technically, as blocks are added into the list
         * of allCollections last. Knowing there is only 9 blocks, and that
         * SUDOKU_SPACES is the same value as the amount of blocks, we can just
         * start our for loop from where the blocks start in this list
         */
        for (int n = allCollections.size() - SUDOKU_SPACES; n < allCollections
                .size(); n++) {
            LinkedList<Cell> currentBlock = allCollections.get(n);
            Cell firstCell = null;
            /*
             * Goes through and looks at all Cells in either a row or
             * a column of the Block, based on the modifier. We know that
             * if we have a list that looks like {0:0,0, 1:0,1, 2:0,2, 3:1,0, 4:1,1..
etc}
             * that in order to get a column from a normal for loop, we
             * need to add 3 onto the first element to get the second Cell
             * in the column. This is used in the if else checks.
             * The for loop itself uses a similar principle, except allowing the
             * for loop to reach '8' for columns, while keeping the limit to 3
             * for the rows modifier(1). This can be tricky to understand at first,
             * attempt to draw the 3x3 grid on paper and work it out manually,
             * inserting the modifier.
             */
            for (int i = 0; i < (SUDOKU_SPACES / modifier); i = i
                    + (BLOCK_SPACES / modifier)) {
                boolean useFirst = true;
                /*
                 * Will get either a the first Cell in the Row or Column,
                 * or will get the second Cell if the first always has a value.
                 * Will never look to the last Cell in a Row or Column as
                 * there would be nothing to compare it to should the first
                 * two be illegal!
                 */
                if (!currentBlock.get(i).hasValue()) {
                    firstCell = currentBlock.get(i);
                    /*
                     * As stated before, this looks for the 'next' row Cell
                     * or column Cell based on the modifier, with the knowledge
                     * that each 'next' column cell is always 3 away from the last,
                     * while each row cell is always 1 away from the last.
                     */
                } else if (!currentBlock.get(i + (1 * modifier)).hasValue()) {
                    firstCell = currentBlock.get(i + (1 * modifier));
                    useFirst = false;
                }
                /*
                 * If a valid cell for pairing was found, send it to the method
                 * to compare with all other cells in the row/column of that
                 * block.
                 */
                if (firstCell != null) {
                    for (int candidate : firstCell.returnCandidates()) {
                        if (pointingPairsPairUp(currentBlock, modifier,
                                candidate, i, useFirst)) {

                            boolean foundCandidate = false;
                            for (Cell cellCheck : currentBlock) {
                                /*
                                 * If the cells match each others pairings, check
                                 * to see if the candidates they have matched up with
                                 * exist in any other Cell in the block, not just in
                                 * the Row or Column. If it does exist in another cell
                                 * outside of the collection, we know that these
                                 * cannot be a pointing pair.
                                 */
                                if (pointingPairsValidateCell(firstCell,
                                        cellCheck, modifier)) {
                                    if (cellCheck.hasCandidate(candidate)
                                            || cellCheck.getValue() == candidate) {
                                        /*
                                         * The 'getValue' check is necessary as
                                         * the current candidate to be removed
                                         * needs to be checked against
                                         * pre-existing values/solved cells in
                                         * the whole collection, not just in the
                                         * block (where it might not exist yet)
                                         * to see if it should not be removed at
                                         * all and move onto  the next candidate value.
                                         */
                                        removeCandidates = false;
                                        foundCandidate = true;
                                    } else {
                                        removeCandidates = true;
                                    }
                                }

                            }
                            /*
                             * Should the checks of removing candidates
                             * and also not finding the candidates in other
                             * cells in the Block pass, then the method may
                             * continue to strip the candidates from other
                             * cells in the row/column that they 'point' to.
                             *
                             * After this, the Result it updated to successful.
                             */
                            if (removeCandidates && !foundCandidate) {
                                if (pointingPairsStripCandidates(modifier,
                                        firstCell, candidate)) {
                                    result.setSuccess(true);
                                    sb = new StringBuffer();
                                    sb.append("<br />Checked for Pointing Pairs (");
                                    sb.append(type);
                                    sb.append(") <br />Removed Candidates<br />");
                                    result.setMessage(sb.toString());
                                    removeCandidates = false;
                                }
                            }
                        }
                    }
                }
            }
        }
        return result;
    }

    /**
     * <p>
     * Part of pointingPairs - removed to break up the code
     * and make it easier to read and maintain.
     * <br />
     * Checks if the two Cell to potentially be paired up
     * in the Row/Column both don't have a value and share the
     * candidate expected.
     * <br />
     * Use first refers to whether the first Cell in the Row/Column
     * was selected, or if the second one was. The if statement
     * handles both of these outcomes with an OR statement, so
     * that if either of these is true, then it may indeed be
     * a pair.
     * </p>
     * @param currentBlock - The block to be checked
     * @param modifier - rows/columns modifier
     * @param candidate - What candidate to be expected and compared
     * @param i - the location of the Cells being compared in the grid, when used
     * with the modifier
     * @param useFirst - whether the first or second Cell is the paired Cell
     * to be checked, based on which was available.
     * @return
     */
    public boolean pointingPairsPairUp(LinkedList<Cell> currentBlock,
            int modifier, int candidate, int i, boolean useFirst) {
        Cell pairCell1 = currentBlock.get(i + (1 * modifier));
        Cell pairCell2 = currentBlock.get(i + (2 * modifier));
        if (((!pairCell1.hasValue() && pairCell1.hasCandidate(candidate)) && useFirst)
                || ((!pairCell2.hasValue()) && (pairCell2
                        .hasCandidate(candidate)))) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * <p>
     * Checks that the Cell being passed does not belong
     * on the same Row or Column as the Pair, ensuring that
     * candidates are not removed from the pair or that
     * candidates are valid (i.e. contained only in that
     * one row/column and not any others in the block).
     * </p>
     * @param firstCell - First cell for pairing check
     * @param cellCheck - Cell to validate whether the
     * candidate is in the single Row/Column or elsewhere
     * in the block.
     * @param modifier - Rows/Columns modifier
     * @return - Whether the validation was true or false.
     */
    public boolean pointingPairsValidateCell(Cell firstCell, Cell cellCheck,
            int modifier) {
        if (((modifier == 1) && cellCheck.getRow() != firstCell.getRow())
                || ((modifier == BLOCK_SPACES) && cellCheck.getColumn() != firstCell
                        .getColumn())) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * <p>
     * In order to split larger chunks of
     * code up from pointingPairs() and make
     * it more maintainable, there is this method
     * that strips the candidates from the other
     * Cells in the Row/Column should the previous
     * tests and validations have passed.
     * </p>
     * @param modifier
     * @param firstCell
     * @param candidate
     * @return
     */
    public boolean pointingPairsStripCandidates(int modifier, Cell firstCell,
            int candidate) {
        boolean success = false;
        /*
         * Based on the modifier, the if statement determines where
         * the row or column is in the list of allCollections.
         * Due to them being added alternatively (Row, Column, Row, Column),
         * it was necessary to use the modifier to add an additional 1 onto
         * the .get command from allCollections, as the Column that might be
         * Column 8, will actually be stored at *2 (as there are Rows there too,
         * doubling the amount of collections), and then +1, next after the Row
         * at location *2.
         */
        if (modifier == 1) {
            collection = allCollections.get(firstCell.getRow() * 2);
        } else {
            collection = allCollections.get((firstCell.getColumn() * 2) + 1);
        }
        for (Cell cell : collection) {
            if (!cell.hasValue()) {
                /*
                 * As long as the Cell in the collection (row/block)
                 * in particular is not in the same block as the
                 * Pointing Pair (so the Pair that cause this event),
                 * remove that candidate as we know it cannot be legal
                 * for this Cell if only the pointing pair can be
                 * this value in their block.
                 */
                if ((cell.getBlock() != firstCell.getBlock())
                        && (cell.hasCandidate(candidate))) {
```

```
                cell.removeCandidate(candidate);
                success = true;
            }
        }
    }
    return success;
}

/**
 * <p>
 * Hidden Singles in Sudoku looks for any cells that have only one candidate
 * left in their candidates list. If they only have one candidate option, it
 * means they must be this candidate without a doubt (presuming correct
 * solutions to this point).
 * </p>
 * 
 * @return - Result of the operation.
 */
public Result hiddenSingles() {
    result = new Result();
    for (int i = 0; i < SUDOKU_SPACES; i++) {
        for (Cell currentCell : allCollections.get(i * 2)) {
            if (!currentCell.hasValue()) {
                if (currentCell.getNumCandidates() == 1) {
                    /*
                     * For all cells in all rows (as what type of collection
                     * doesn't matter as this is solved on a cell to cell
                     * basis) check if there is only one option for a cell
                     * to be out of their candidates. If yes, set it to this
                     * value.
                     */
                    currentCell.assignOnlyCandidate();
                    result.setSuccess(true);
                    result.addAffected(currentCell);
                    result.setMessage("<br />Checked for Hidden Singles<br />");
                }
            }
        }
    }
    return result;
}

/**
 * <p>
 * Checks all Cells in the puzzle. If each one of them has a value (based on
 * a counter increasing on each positive value found, then the method
 * returns true. <br />
 * MAX_CELLS = 81, the full Sudoku grid cells.
 * </p>
 * 
 * @return boolean referring to is the puzzle has been solved or not.
 */
public boolean solved() {
    int solvedCells = 0;
    for (int i = 0; i < SUDOKU_SPACES; i++) {
        for (int j = 0; j < SUDOKU_SPACES; j++) {
            if (grid[i][j].hasValue()) {
                solvedCells++;
            }
        }
    }
    if (solvedCells == MAX_CELLS) {
        return true;
    } else {
        return false;
    }
}

// ---------------------------------------------------------

/**
 * @return the current full grid of Cells.
 */
public Cell[][] getGrid() {
    return grid;
}

/**
 * <p>
 * Appends common information to a StringBuffer to be set and returned in
 * the result. In particular, information concerning cells and their
 * location.
 * </p>
 *
 * @param cell
 *            - current cell modified
 * @return the completed and concatanated String
 */
public String appendCellInfo(Cell cell) {
    sb.append("<br />Updated: ");
    sb.append(cell.getRow() + 1);
    sb.append(", ");
    sb.append(cell.getColumn() + 1);
    sb.append("<br /");
    return sb.toString();
}

/**
 * <p>
 * My original way of viewing the steps and results taken as the Sudoku
 * puzzle solves. Left in for sake of re-usability should the GUI need to be
 * removed. <br />
 * Prints out the current status of the board, with dividers for the Blocks
 * and underscores for unsolved cells. Does not display candidates.
 * </p>
 */
public void printLines() {
    for (int i = 0; i < SUDOKU_SPACES; i++) {
        if ((i + 1 / BLOCK_SPACES) % BLOCK_SPACES == 0) {
            for (int k = 0; k < 7; k++) {
                System.out.print("_ ");
            }
            System.out.println();
        }
        for (int j = 0; j < SUDOKU_SPACES; j++) {
            if ((j + 1 / BLOCK_SPACES) % BLOCK_SPACES == 0) {
                System.out.print('|');
            }
            if (grid[i][j].getValue() == 0) {
                System.out.print('-');
            } else {
                System.out.print(grid[i][j].getValue());
            }
        }
```

```
            System.out.print("| \n");
        }

    }

}
```

Cell:

```
package assignment1;

import java.util.LinkedList;

/**
 * <h2>Cell</h2>
 * <p>
 * Cell class holds all information needed for this implementation of the
 * SudokuSolver problem.<br />
 * The Cell holds data about it's position on the grid (row, column), what block
 * it is in, what candidates it has got, whether it was a value passed in on the
 * first setup of the grid or a solved Cell and any value it may contain (0-9).
 * </p>
 * <p>
 * It's methods deal with assigning values, such as its respective value and
 * attributes connected to its location on the grid and removing candidates. <br
 * />
 * <p>
 * There are also a number of methods for other classes to call upon to get
 * answers about the cells current status, such as whether it has a candidate,
 * what candidates it has, whether it has a value and if it is the same (in the
 * same location) as another Cell.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class Cell {

    private int value = 0;
    private int block = 0;
    private int row = 0;
    private int column = 0;
    private int[] candidates;
    private boolean firstValue;

    /**
     * <p>
     * The constructor is passed parameters of the potential value of the cell
     * and which row and column it belongs to on the grid.
     * </p>
     * <p>
     * From this information, the cell will either be set up with a list of
     * candidates (1 - 9) or will assign the value passed if given a legal
     * value. If a value is set in this way, it is also noted for use in the GUI
     * and setting the colour of all original values to different than that of
     * the solved Cells.
     * </p>
     *
     * @param num
     * @param row
     * @param column
     */
```

```
public Cell(int num, int row, int column) {
    candidates = new int[9];
    this.row = row;
    this.column = column;

    if (num < 1 || num > 10) {
        value = 0;
        for (int i = 0; i < 9; i++) {
            candidates[i] = (i + 1);
        }
    } else {
        value = num;
        firstValue = true;
    }

}

/**
 * <p>
 * Takes in a number to be removed from the list of candidates, checks that
 * the value is legal and then removes it from the correct element of the
 * array (- 1 added to remove the correct value with arrays beginning at 0
 * for candidate 1.
 * </p>
 *
 * @param candidate
 *            - to be removed from the list
 */
public void removeCandidate(int candidate) {
    if (this.testBounds(candidate)) {
        candidates[candidate - 1] = 0;
    }
}

/**
 * <p>
 * Goes through the list of candidates in the candidates array and adds them
 * to a LinkedList if the number's value is higher than 0, i.e. exists as a
 * valid candidiate.
 * </p>
 *
 * @return toReturn - a list of the candidates, stripped of any 0's in the
 *         array they are held in here.
 */
public LinkedList<Integer> returnCandidates() {
    LinkedList<Integer> toReturn = new LinkedList<Integer>();
    for (int i = 0; i < candidates.length; i++) {
        if (candidates[i] > 0) {
            toReturn.add(candidates[i]);
        }
    }
    return toReturn;
}

/**
 * <p>
 * A method that compares and counts the amount of matching candidates in
 * the candidates list here in the cell and the passed list by looking
 * through the array and finding numbers with a value higher than 0 and
 * incrementing a counter on each match.
 * </p>
 *
 * @param toMatch
 *            - List with values to compare to the candidates in the cell.
 * @return amount - The amount of matches between the passed candidates and
 *         the candidates in this cell.
 */
public int matchingCandidates(LinkedList<Integer> toMatch) {
    int amount = 0;
    for (int check : toMatch) {
        if (check > 0) {
            if (this.hasCandidate(check)) {
                amount++;
            }
        }
    }
    return amount;
}

/**
 * <p>
 * On request of this method, the list of candidates is iterated through and
 * the value found above 0 is applied to be the value of this cell. <br />
 * Checks that there is only one candidate before applying the core
 * function.
 * </p>
 */
public void assignOnlyCandidate() {
    if (this.getNumCandidates() == 1) {
        for (int i = 0; i < candidates.length; i++) {
            if (candidates[i] != 0) {
                this.setValue(candidates[i]);
            }
        }
    }
}

/**
 * <p>
 * Sets all candidates in the array to '0', where '0' represents 'nothing'.
 * </p>
 */
public void clearCandidates() {
    for (int i = 0; i < 9; i++) {
        candidates[i] = 0;
    }
}

/**
 * <p>
 * First checks to see if the value passed would be out of the array bounds
 * of the cell. If the test is passed, the value is passed to the array with
 * -1 to check the element location where the value would be. If the value
 * matches the expected value (e.g. element 5, value 6, given value 6, then
 * the method returns true.
 * </p>
 *
 * @param candidate
 *            - Number to check
 * @return - Returns a boolean whether the candidate exists in this cell or
 *         not.
 */
public boolean hasCandidate(int candidate) {
    if (this.testBounds(candidate)) {
        if (candidates[candidate - 1] == candidate) {
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}

/**
 * <p>
 * Checks through the array of candidates and checks to see which values are
 * above 0. Any above 0 are considered a valid candidate and a counter is
 * incremented to represent this. Once the iteration is completed, the
 * method returns how many candidates it found.
 * </p>
 *
 * @return numCandidates - The amount of candidates contained in the array
 *         of candidates.
 */
public int getNumCandidates() {
    int numCandidates = 0;
    for (int candidate : candidates) {
        if (candidate != 0) {
            numCandidates++;
        }
    }
    return numCandidates;
}

/**
 * <p>
 * Checks if the number in the variable 'value' is not 0. Logically, if it
 * is not 0, it should be a valid value. <br />
 * The method returns whether this value is 0 or the cell contains an actual
 * value.
 * </p>
 *
 * @return boolean value of if the cell has a value or not
 */
public boolean hasValue() {
    if (value != 0) {
        return true;
    }
    return false;
}

/**
 * <p>
 * Sets the passed value as the value for this cell and then clears the
 * candidate list to be sure that any other class later requesting
 * candidates doesn't confuse this cell with an assigned value with an
 * unsolved cell.
 * </p>
 *
 * @param value
 *            - The 'solved' value for the cell to hold
 */
public void setValue(int value) {
    this.value = value;
    this.clearCandidates();
```

```
}

/**
 * @return the cell value.
 */
public int getValue() {
    return value;
}

/**
 * @return the row the cell is on.
 */
public int getRow() {
    return row;
}

/**
 * @return the column the cell is on.
 */
public int getColumn() {
    return column;
}

/**
 * @return the block the cell belongs to.
 */
public int getBlock() {
    return block;
}

/**
 * <p>
 * Sets the block the cell belongs to from the passed integer.
 * </p>
 *
 * @param block
 *            the block this cell should belong to.
 */
public void setBlock(int block) {
    this.block = block;
}

/**
 * @return boolean whether this cell was assigned on the inital read in of
 *         the grid or a later solved cell.
 */
public boolean firstValue() {
    return firstValue;
}

/**
 * <p>
 * Tests whether a value is valid (between 0 and up to and including 9).
 * </p>
 *
 * @param num
 *            - passed value to check
 * @return boolean if the value passed the test.
 */
public boolean testBounds(int num) {
    if (num < 10 && num > 0) {
        return true;
        } else {
            return false;
        }
    }

    /**
     * Overwritten equals method: Implementation only requires that the Cells in
     * comparison would occupy the same row/column to be known as 'equal', i.e.
     * The same Cell.
     *
     * @see java.lang.Object#equals(java.lang.Object)
     *
     */
    @Override
    public boolean equals(Object checker) {
        Cell cellCheck = (Cell) checker;
        if (cellCheck.getRow() == row && cellCheck.getColumn() == column) {
            return true;
        } else {
            return false;
        }
    }
}
```

Result:

```
package assignment1;

import java.util.LinkedList;

/**
 * <h2>Result</h2>
 * <p>
 * Rather than simply returning a boolean value of whether a solving algorithm
 * was correct or not, I wished to return information about which Cell(s) was
 * altered too, in order to be demonstrated on the grid. <br />
 * This class holds values of the success, a message about the task completed
 * and any Cells affected.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class Result {

    private boolean success;
    private String message;
    private LinkedList<Cell> affected;

    /**
     * <p>
     * Setup of Result to stop NullPointers
     * </p>
     */
    public Result() {
        success = false;
        message = " ";
        affected = new LinkedList<Cell>();
    }

    /**
     * @param re
     *             - Set result of operation.
     */
    public void setSuccess(boolean re) {
        success = re;
    }

    /**
     * @return - Return result of operation.
     */
    public boolean getSuccess() {
        return success;
    }

    /**
     * @param msg
     *             - The message associated with the outcome of the operation to
     *             be set.
     */
    public void setMessage(String msg) {
        message = msg;
    }

    /**
     * @return - The message associated with the operation carried out, to be
     *         returned for display to the user.
     */
    public String getMessage() {
        return message;
    }

    /**
     * @param cell
     *             - Cell to be added to the list of Cells affected by the
     *             solving algorithm and step this Result is associated with.
     */
    public void addAffected(Cell cell) {
        affected.add(cell);
    }

    /**
     * @return The list of Cells affected by the solving algorithm that created
     *         this Result.
     */
    public LinkedList<Cell> getAffected() {
        return affected;
    }

}
```

SolverFrame:

```
package assignment1;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;

import javax.swing.JButton;
```

```java
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.KeyStroke;
import javax.swing.border.LineBorder;

/**
 * <h2>SolverFrame</h2>
 * <p>
 * The main visual displayer for this application.<br />
 * One Frame to rule them all, One Frame to find them,<br />
 * One Frame to bring them all and in the darkness bind them. <br />
 * This class holds reference to the Model and is used by the Controller
 * (Driver). Most actions by the user are caught and passed to the Model from
 * here.
 * </p>
 * <p>
 * From the frame, the user can open files, close the application, take 'steps'
 * through a Sudoku's solving process and see a quick solve. The steps taken
 * towards the solution are also displayed, although where they were executed is
 * not. <br />
 * The GUI is very simple. Were there more time alloted for this project, I
 * would expand into firstly using JTextArea and carats to get the Solution
 * steps to autoscroll to the bottom of the steps. <br />
 * Next I would get a log of not just what steps were taken but -where- each
 * step was taken in the grid puzzle. Right now the user must see by eye what
 * has taken place. Once this is implemented, I would work towards allowing the
 * user to step take a step backwards through the puzzle. <br />
 * Were much more time alloted, I would expand into allowing the user to click
 * buttons in attempts to solve the puzzle by themselves, either through given
 * methods or directly allowing them attempts at solving the puzzle with the
 * methods just used to confirm correct choices.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
@SuppressWarnings("serial")
public class SolverFrame extends JFrame implements ActionListener {

    private SolverCanvas canvas;
    private boolean gridLoaded = false;
    private boolean threadRunning = false;
    private Thread runSolver;
    private String appTitle = "Sudoku Solver - jee22";

    // === Menu Items ===
    private JMenuBar menuBar;
    private JMenu fileMenu;
    private JMenuItem openItem;
    private JMenuItem exitItem;
    private JFileChooser fileChooser;

    // === Sidebar Items ===
    private JPanel sidebar;
    private JButton button;
    private JButton solveButton;
    private JPanel stepsPanel;
    private JScrollPane scroll;
    private JLabel status;
    private JLabel stText;

    /**
     * <p>
     * The constructor sets it's own components and prepares the canvas where
     * the Sudoku grid is displayed to be ready for use too.
     * </p>
     *
     * @param solver
     *            - reference to an opening SudokuSolver to get the application
     *            running.
     */
    public SolverFrame(SudokuSolver solver) {
        canvas = new SolverCanvas(solver);
        setupFrameProperties();
        setupMenu();
        menuBar.add(fileMenu);

        setupSidebar();
        this.add(sidebar, BorderLayout.EAST);
        this.getContentPane().add(canvas, BorderLayout.CENTER);
        this.setJMenuBar(menuBar);
    }

    /**
     * <p>
     * Assigns the properties of this Frame and how it should be displayed to a
     * user.
     * </p>
     * <p>
     * In particular, the grid is set to a specific size and is unsizable in
     * order to keep the Sudoku grid displaying correctly with uniformly sized
     * cells. I felt this was the best choice for such a simple GUI and
     * application at this point in time. <br />
     * The Frame is created in the centre of the users screen and set to 'grey',
     * the default of most OS general application colours.
     * </p>
     */
    public void setupFrameProperties() {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setMinimumSize(new Dimension(700, 550));
        this.pack();
        this.setResizable(false);
        this.setTitle(appTitle);
        this.setBackground(Color.gray);
        this.setLocationRelativeTo(null);
        this.setVisible(true);
    }

    /**
     * <p>
     * Establishes the attributes of the top bar menu on the frame, allowing the
     * user to open files and exit the application.
     * </p>
     * <p>
     * Each button the menu bar is given a quick key shortcut for ease of use
     * and also a command for use with the ActionListener that listens for uses
     * by the user. <br />
     * A small description of each action is given for accessibility sake of
     * sight impaired users.
     * </p>
     */
    public void setupMenu() {
        menuBar = new JMenuBar();
        fileMenu = new JMenu("File");
        openItem = new JMenuItem("Open");
        exitItem = new JMenuItem("Exit");

        openItem = new JMenuItem("Open", KeyEvent.VK_O);
        openItem.setActionCommand("open");
        openItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O, Toolkit
                .getDefaultToolkit().getMenuShortcutKeyMask()));
        openItem.getAccessibleContext().setAccessibleDescription(
                "Opens a Sudoku .sud file");
        fileMenu.add(openItem);

        exitItem = new JMenuItem("Exit");
        exitItem.setActionCommand("exit");
        exitItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Q, Toolkit
                .getDefaultToolkit().getMenuShortcutKeyMask()));
        exitItem.getAccessibleContext().setAccessibleDescription(
                "Exit Sudoku Solver");
        fileMenu.add(exitItem);

        fileMenu.add(openItem);
        fileMenu.add(exitItem);

        openItem.addActionListener(this);
        exitItem.addActionListener(this);
    }

    /**
     * <p>
     * Sets the sidebar up for use on the Frame.<br />
     * Strict dimensions are set to ensure it doesn't tamper with the Sudoku
     * grid display.
     * </p>
     * <p>
     * To give the user some help with the application, there are labels that
     * display the currently open file, buttons to either take steps in solving
     * the puzzle or a button to auto solve and pause the puzzle. <br />
     * As previously stated,were there more time I would work on making a better
     * JScrollBar implementation.
     * </p>
     */
    public void setupSidebar() {
        sidebar = new JPanel();
        sidebar.setPreferredSize(new Dimension(200, 500));

        stText = new JLabel("File Open: ");
        stText.setPreferredSize(new Dimension(55, 10));
        status = new JLabel("No file Open");
        status.setPreferredSize(new Dimension(135, 10));

        sidebar.add(stText);
        sidebar.add(status);

        button = new JButton("Take Step");
        button.addActionListener(this);
        button.setActionCommand("step");
        sidebar.add(button);
```

```
        solveButton = new JButton("Solve Puzzle");
        solveButton.addActionListener(this);
        solveButton.setActionCommand("solve");
        sidebar.add(solveButton);

        stepsPanel = new JPanel();
        stepsPanel.setLayout(new BorderLayout());
        stepsPanel.setBackground(Color.WHITE);
        stepsPanel.setBorder(new LineBorder(Color.BLACK));

        scroll = new JScrollPane(stepsPanel);
        scroll.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        scroll.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        scroll.setPreferredSize(new Dimension(180, 400));

        sidebar.add(scroll);

    }

    /**
     * <p>
     * Commands sent by the user from menus and buttons.
     * </p>
     * <p>
     * The open command will stop any running Thread to ensure that if a puzzle
     * is being solved, the user can't open a new puzzle and have the solving
     * continue on the new puzzle too.
     * </p>
     */
    @Override
    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        if (command.equals("open")) {
            if (threadRunning) {
                threadHandler();
            }
            if (openFile()) {
                stepsPanel.removeAll();
                repaint();
                gridLoaded = true;
            }
        }
        if (command.equals("exit")) {
            exit();
        }
        if (command.equals("step")) {
            if (gridLoaded) {
                if (threadRunning) {
                    threadHandler();
                }
                canvas.takeStep();
                stepsPanel.add(canvas.getSteps());
                this.repaint();
            } else {
                // Pop up box to say no sudoku loaded - Future implementation
            }

        }
        /**
         * <p>
         * If the user wishes the solve the puzzle, it is first checked if there
         * is a loaded grid, then if the puzzle is being solved, and if so stops
         * it, then starts a new Thread and begins running a new solve loop in
         * SudokuCanvas. <br />
         * The Solve button is also changed to 'Stop Solving', useful for
         * pausing a solve mid way through.
         * </p>
         */
        if (command.equals("solve")) {
            if (gridLoaded) {
                if (threadRunning) {
                    this.threadHandler();
                } else {
                    if (!canvas.isSolved()) {
                        runSolver = new Thread(canvas);
                        threadRunning = true;
                        runSolver.start();
                        stepsPanel.add(canvas.getSteps());
                        this.repaint();
                        solveButton.setText("Stop Solving");
                    }
                }
            } else {
                // Pop up box to say no sudoku loaded
            }
        }
    }

    /**
     * <p>
     * If there is a thread running, often methods that would be otherwise
     * affected by it's continuation will call this function. <br />
     * This method calls to the SudokuCanvas to set a boolean value to false in
     * order to stop a while loop in the canvas, then interrupts the Thread from
     * here. <br />
     * Once done, the boolean keeping track of if a Thread is running is set to
     * false and the button stopping/starting the Thread/solver displays a
     * relevant message.
     * </p>
     */
    public void threadHandler() {
        canvas.pause();
        runSolver.interrupt();
        threadRunning = false;
        solveButton.setText("Solve Puzzle");
    }

    /**
     * <p>
     * Opens a file on the users system to solve
     * </p>
     * <p>
     * Uses <code>JFileChooser</code> to allow the user to select any file in
     * their system.
     * </p>
     * <p>
     * In order to filter out invalid files, I have created a customer
     * <code>FileFilter</code> that only displays <code>.sud</code> files.
     * </p>
     * <p>
     * Once a users has selected a file, it passes it through onto the canvas to
     * open and be displayed.
     * </p>
```

```
     */
    private boolean openFile() {
        fileChooser = new JFileChooser();
        fileChooser.setFileFilter(new SudFileFilter());
        int chosen = fileChooser.showOpenDialog(this);
        if (chosen == JFileChooser.APPROVE_OPTION) {
            status.setText(fileChooser.getName(fileChooser.getSelectedFile()));
            canvas.openFile(fileChooser.getSelectedFile());
            return true;
        } else {
            return false;
        }
    }

    /**
     * <p>
     * Exits the program.
     * </p>
     */
    private void exit() {
        System.exit(0);
    }
}
```

SolverCanvas:

```
package assignment1;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.GridLayout;
import java.io.File;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingConstants;
import javax.swing.border.LineBorder;

/**
 * <h2>SudokuCanvas</h2>
 * <p>
 * The visual JPanel that holds the Cells of the Sudoku puzzle grid and displays
 * them and their status to the user.
 * </p>
 * <p>
 * This class is controlled by the SolverFrame, and holds reference to the
 * SudokuSolver, allowing it to get data directly from the Solver, and pass on
 * requests from the SudokuFrame when buttons to take solving steps are clicked.
 * </p>
 * <p>
 * Given more time I would work more on this section of the application,
 * displaying references to each cell and assigning them 'names', e.g. A1, A2,
 * C4, etc. Due to time constraints, I kept the canvas as simple yet informative
 * of the solving steps taking place as I could.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
@SuppressWarnings("serial")
public class SolverCanvas extends JPanel implements Runnable {
```

```
private SudokuSolver solver;
private FileHandler fileH;
private boolean gridLoaded = false;
private boolean keepSolving = false;
private Result result;

// ===== GUI ELEMENTS =====
private JLabel cellLabel;
private JLabel[][] cells;
private String textValue;
private Font lFont;
private Font sFont;
private JLabel stepsLabel;
private int largeText = 30;
private int smallText = 12;

private Cell[][] grid;
private StringBuffer sb;
private StringBuffer stepB;

private static final int SUDOKU_SPACES = 9;

/**
 * <p>
 * Constructor creates instances of variables needed to allow the 'canvas'
 * piece to function and display correctly.
 * </p>
 *
 * @param solver
 *            - a reference of the solver created on the applications start
 *            up.
 */
SolverCanvas(SudokuSolver solver) {
    this.solver = solver;
    cells = new JLabel[SUDOKU_SPACES][SUDOKU_SPACES];
    lFont = new Font("Arial", Font.BOLD, largeText);
    sFont = new Font("Arial", Font.PLAIN, smallText);
    stepsLabel = new JLabel();
    stepsLabel.setVerticalAlignment(SwingConstants.TOP);
    stepsLabel.setLayout(new GridLayout());
    stepsLabel.setVerticalAlignment(SwingConstants.TOP);
    stepsLabel.setLayout(new GridLayout());
    setupCanvas();
}

/**
 * <p>
 * Sets up various things for this canvas, in order to help display the
 * actions and steps being taken in solving the problem.
 * </p>
 * <p>
 * In particular, the background is set to White to better display darker
 * coloured cells, and a GridLayout is used as it ensures uniform size to
 * each of the elements contained. In particular, the only things contained
 * are the 'cellLabel's, one for each Cell of the Sudoku grid. <br />
 * Each cellLabel is bordered to help define it's own bounds. To further the
 * usefulness of the display, the 3x3 'blocks' of the grid are displayed
 * with alternating colours (Gray, light gray), using an if statement to
 * determine whether the current cell is within a 'gray' block or otherwise.
 * <br />
 * This is discovered through use of the same nested for loop that is used
 * to create new cellLabels, up to the amount of Sudoku Cells.
 * </p>
 * <p>
 * Once the 'cells' are set up to be displayed, they are added to a 2D array
 * to keep hold of them and their respective locations in parallel to those
 * used in the SudokuModel class.
 * </p>
 */
public void setupCanvas() {
    this.setBackground(Color.WHITE);
    this.setLayout(new GridLayout(SUDOKU_SPACES, SUDOKU_SPACES));
    for (int i = 0; i < SUDOKU_SPACES; i++) {
        for (int j = 0; j < SUDOKU_SPACES; j++) {
            cellLabel = new JLabel();
            cellLabel.setBorder(new LineBorder(Color.BLACK));
            cellLabel.setOpaque(true);

            colourCell(i, j, cellLabel);

            this.add(cellLabel);
            cells[i][j] = cellLabel;
        }
    }
}

/**
 * <p>
 * When passed a file, will create a new instance of the FileHandler class,
 * clear all Cells from the current state of this class and re-setup all for
 * a fresh 'canvas', read in the file and set the correct grid and Cells
 * based on the data read in and returned, calls for the Solver to setup
 * what it must and then requests from the Solver to get a copy of the grid
 * once setup to be displayed to the user. <br />
 * gridLoaded keeps the class aware of if there is currently a grid on
 * display or not, used mainly for the first time the application is opened,
 * to keep all squares blank before a grid is loaded.
 * </p>
 *
 * @param file
 *            - The file to be opened by the FileHandler.
 */
public void openFile(File file) {
    fileH = new FileHandler();
    this.removeAll();
    setupCanvas();
    solver = new SudokuSolver();
    solver.setGrid(fileH.readFile(file));
    solver.setup();
    grid = solver.getGrid();
    gridLoaded = true;
    this.repaint();
}

@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    grid = solver.getGrid();
    drawgrid(g);
}

/**
 * <p>
 * Draws/Writes onto the JLabels representing the Cells, contained within a
```

```
     * 2D array, based upon the data in the grid of SudokuSolver, to determine
     * if a Cell needs to display a start value, a solved value or a list of
     * current candidates for that Cell.
     * </p>
     * <p>
     * In order to get wrapping text, HTML tags are used that keep the text
     * contained. <br />
     * As the method loops through the 2D array of cells, it checks if the Cell
     * has a value or not. Should the cell have a value, the text is Centred to
     * the JLabel and the text is updated to be just the value of the cell.
     * Should this value be taken on the read in, it is set to display as Blue.
     * If the cell is solved and updated, it is displayed as Red.
     * </p>
     * <p>
     * If there is not a value to the cell, then the text is set to be smaller
     * and aligned to the top left of the JLabel, in order to neatly display the
     * candidates of the cell. <br />
     * In order to display the candidates in a pleasing manner, the method uses
     * a StringBuffer, only taking in the candidates (not the [ and ] of the
     * array], and appending spaces between the candidates.
     * </p>
     * <p>
     * Also gets the latest results from the last successful algorithm
     * operation. If there are any Cells in the affected list, the JLable
     * associated with them has its background set to Yellow, to make it visible
     * to the user which cells were updated. Given more time on this project, I
     * would work on getting a text explanation and description about these
     * updates (and the candidates) in the sidebar.
     * </p>
     *
     * @param g
     *            - Graphics to be drawn, passed by 'paintComponent'.
     */
    private void drawgrid(Graphics g) {
        if (gridLoaded) {
            for (int i = 0; i < SUDOKU_SPACES; i++) {
                for (int j = 0; j < SUDOKU_SPACES; j++) {
                    colourCell(i, j, cells[i][j]);
                    sb = new StringBuffer();
                    sb.append("<html>");
                    if (grid[i][j].hasValue()) {
                        cells[i][j]
                                .setHorizontalAlignment(SwingConstants.CENTER);
                        cells[i][j].setVerticalAlignment(SwingConstants.CENTER);
                        textValue = Integer.toString(grid[i][j].getValue());
                        cells[i][j].setFont(lFont);
                        if (grid[i][j].firstValue()) {
                            cells[i][j].setForeground(Color.BLUE);
                        } else {
                            cells[i][j].setForeground(Color.RED);
                        }
                    } else {

                        for (int candidate : grid[i][j].returnCandidates()) {
                            sb.append(candidate);
                            sb.append(" ");
                        }
                        textValue = sb.toString();
                        cells[i][j].setVerticalAlignment(SwingConstants.TOP);
                        cells[i][j].setFont(sFont);
                    }
                    sb.append("</html>");
                    cells[i][j].setText(textValue);
                    result = solver.returnResult();
                    if (!solver.isSolved()) {
                        for (Cell rC : result.getAffected()) {
                            cells[rC.getRow()][rC.getColumn()]
                                    .setBackground(Color.YELLOW);
                        }
                    }
                }
            }
        }
    }

    public void colourCell(int i, int j, JLabel cellAtm) {
        if ((j > 2 && j < 6) && (i < 3 || i > 5)
                || ((j < 3 || j > 5) && (i > 2 && i < 6))) {
            cellAtm.setBackground(Color.LIGHT_GRAY);
        } else {
            cellAtm.setBackground(Color.GRAY);
        }
    }

    /**
     * <p>
     * Creates a new instance of StringBuffer to the same location as the one
     * also used for labelling the JLabels. <br />
     * This is used to wrap results from the SudokuSolvers steps taken in HTML.
     * <br />
     * The request to the SudokuSolver to take a step forward is sent and will
     * return a String for using in displaying on the main Frame's sidebar. <br />
     * Repaint is called at the end of the method to ensure all visuals are
     * updated.
     * </p>
     */
    public void takeStep() {
        stepB = new StringBuffer();
        stepB.append("<html>");
        stepB.append(solver.takeStep());
        stepB.append("</html>");
        stepsLabel.setText(stepB.toString());
        this.repaint();
    }

    /**
     * <p>
     * Returns a JLabel of text describing what steps were taken so far, and how
     * many.
     * </p>
     *
     * @return the Text built up from the 'takeStep()' method for use in the
     *         SudokuFrame.
     */
    public JLabel getSteps() {
        return stepsLabel;
    }

    /**
     * <p>
     * Implementation of the run() method, calling while the puzzle is not
     * solved, continue taking steps and adding them to the Label that takes
     * care of recording what steps have been taken so far.
     * </p>
```

```
     * <p>
     * Thread.sleep is used to allow pauses between steps taken in the solving
     * of the puzzle, giving the user time to view the steps taken even with the
     * puzzle is being auto-solved.
     * </p>
     */
    @Override
    public void run() {
        keepSolving = !solver.isSolved();
        while (!solver.isSolved() && keepSolving) {
            stepB = new StringBuffer();
            stepB.append("<html>");
            stepB.append(solver.takeStep());
            stepB.append("</html>");
            stepsLabel.setText(stepB.toString());
            this.repaint();
            try {
                Thread.sleep(150);
            } catch (InterruptedException e) {
                // An interrupted Exception is thrown here if the Solve button
                // is
                // pressed repeatedly without first solving the puzzle.
                // This is due to the thread being interrupted during the
                // 'sleep'
                // Since the thread is correctly stopped and I am aware of this
                // issue, I have not printed it out to the command line or
                // further handled the Exception. I am aware this is not ideal
                // practice, however, given the time constraints, I would deal
                // with this on high priority if given more time to work on the
                // GUI rather than the solving algorithms.
            }
        }
        keepSolving = false;
    }

    /**
     * <p>
     * Helps stopping the program by setting boolean allowing the while loop to
     * continue to false.
     * </p>
     */
    public void pause() {
        keepSolving = false;
    }

    /**
     * @return If puzzle is solved or not
     */
    public boolean isSolved() {
        return solver.isSolved();
    }
}
```

FileHandler:

```
package assignment1;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
```

```java
/**
 * <h2>FileHandler</h2>
 * <p>
 * A custom <code>FileFilter</code> class to only show a user <code>.sud</code>
 * files when they open <code>JFileChooser</code> to load a file on their
 * system. <br />
 * Will still allow the use of 'All files' option, however. Needs reworking.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class FileHandler {

    private BufferedReader buffReader;
    private static final int SUDOKU_SPACES = 9;
    private char[][] lines;

    public FileHandler() {
        lines = new char[SUDOKU_SPACES][SUDOKU_SPACES];
    }

    public FileHandler(String fileName) {
        lines = new char[SUDOKU_SPACES][SUDOKU_SPACES];
        lines = readFile(newFile(fileName));
    }

    /**
     * <p>
     * Allows the creating of a new File object based on a file's name, assuming
     * it's location is local to the application source files.
     * </p>
     *
     * @param fileName
     *            Takes the file name of the File to be made
     * @return returns the reference to the File.
     */
    public File newFile(String fileName) {
        return new File(fileName);
    }

    /**
     * <p>
     * Takes in a File object and opens the file, reading in the data and
     * putting it into first Strings, then getting an array of Characters to be
     * used as the grid for the Sudoku puzzle grid. <br />
     * Contains a number of Exception catches. Should I have had more time on
     * these project, handling of incorrect data types, files and locations
     * would be made more rigorous.
     * </p>
     *
     * @param file
     * @return
     */
    public char[][] readFile(File file) {
        try {
            buffReader = new BufferedReader(new FileReader(file));

            for (int i = 0; i < SUDOKU_SPACES; i++) {
                String singleLine = buffReader.readLine();
                lines[i] = singleLine.toCharArray();
            }

            buffReader.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (NullPointerException e) {
            /*
             * Catches a NullPointerException should the file be an incorrect
             * format - .sud
             */
            e.printStackTrace();
        }
        return lines;
    }
}
```

SudFileFilter:

```
package assignment1;

import java.io.File;
import javax.swing.filechooser.FileFilter;

/**
 * <h2>SudFileFilter</h2>
 * <p>
 * A custom <code>FileFilter</code> class to only show a user <code>.sud</code>
 * files when they open <code>JFileChooser</code> to load a file on their
 * system.
 * </p>
 *
 * @author James Euesden - jee22@aber.ac.uk
 * @version 1.0
 */
public class SudFileFilter extends FileFilter {

    /**
     * <p>
     * <code>return true</code> is checking each path and file, and if they end
     * with <code>.sud</code> or are a Directory file, they will be displayed.
     * </p>
     */
    @Override
    public boolean accept(final File path) {
        if (path.isDirectory()) { // Returns true and displays Folders
            return true;
        }
        String name = path.getName().toLowerCase();
        if (name.endsWith("sud")) { // Returns true and displays any file ending
                                    // .sud
            return true;
        }
        return false;
    }

    @Override
    public String getDescription() {
        return null;
    }
}
```