**Seyed Ali Mojarrad**

**Comp 282 – Mon/Wed**

**Assignment 1 – Fixed**

**02/23/2015**

**Source Code:**

/\* Ali Mojarrad

\* Comp282 - Mon/Wed

\* Assignment 1 - FIXED

\* 02/23/2015

\* This is the Fixed version of Sudoku assignment to solve Sudoku puzzles \*/

**class** Spot {

**private** **int** row, col;

// Constructor

**public** Spot(**int** row, **int** col) {

**this**.row = row;

**this**.col = col;

}

**public** **void** setRow(**int** row) {

**this**.row = row;

}

**public** **void** setCol(**int** col) {

**this**.col = col;

}

**public** **int** getRow() {

**return** row;

}

**public** **int** getCol() {

**return** col;

}

}

**class** sudoku {

**private** **int** board[][];

// default constructor - creates an empty 9\*9 board

**public** sudoku() {

board = **new** **int**[9][9];

**for** (**int** i = 0; i < 9; i++)

**for** (**int** j = 0; j < 9; j++)

board[i][j] = 0;

}

// Construct a new sudoku puzzle from a string

// This piece of code might be useful to you:

// (int) (s[row].charAt(col + col/3)) - 48

**public** sudoku(String s[]) {

// create a 9\*9 board - could've used default constructor but preferred this way

board = **new** **int**[9][9];

// fill the board from given string

**for** (**int** iRow = 0; iRow < 9; iRow++)

**for** (**int** jCol = 0; jCol < 9; jCol++)

board[iRow][jCol] = (**int**) (s[iRow].charAt(jCol + jCol / 3) - 48);

}

// returns our board (Getter)

**public** **int**[][] getBoard() {

**return** board;

}

// Copy constructor

**public** sudoku(sudoku p) {

board = **new** **int**[9][9];

**for** (**int** iRow = 0; iRow < 9; iRow++)

**for** (**int** jCol = 0; jCol < 9; jCol++)

board[iRow][jCol] = (**int**) (p.board[iRow][jCol]);

}

// Hint: use String.valueOf( i ) to convert an int to a String

**public** String toString() {

String output = **new** String();

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

output += String.*valueOf*(board[iRow][jCol]);

// add | after every third column between numbers

**if** ((jCol ) % 3 == 0 && jCol != 8) {

output += " | ";

}

}

output += "\n";

// add | after every third row between numbers

**if** ((iRow ) % 3 == 0 && iRow != 8) {

output += "---------------\n";

}

}

**return** output;

}

// for easy checking of your answers

**public** String toString2() {

String result = **new** String();

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

result = result + String.*valueOf*(board[iRow][jCol]);

}

}

**return** result;

}

// create rotated sudoku puzzle – used by my test programs

**public** **void** rotate() {

**int**[][] temp = **new** **int**[9][9];

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

temp[jCol][8 - iRow] = board[iRow][jCol];

}

}

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

board[iRow][jCol] = temp[iRow][jCol];

}

}

}

// Does the current board satisfy all the sudoku rules?

**public** **boolean** isValid() {

**boolean** valid = **true**;

// Go through entire board

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

**int** num = board[iRow][jCol];

// have 3 counts for checking

**int** count = 0;

**int** count1 = 0;

**int** count2 = 0;

// when num is not 0

**if** (num != 0) {

/\* simple loop for scanning the board \*/

**for** (**int** k = 0; k < 9; k++) {

// if the spot has the number in it same row/all col

**if** (board[iRow][k] == num) {

// increment count

count++;

}

// if count is more than one then return it false

**if** (count > 1) {

valid = **false**;

}

// same as above but alternate rows/ same col

**if** (board[k][jCol] == num) {

count1++;

}

**if** (count1 > 1) {

valid = **false**;

}

}

// find the top left spot for the boxes

**int** rowStart = (iRow / 3) \* 3;

**int** colStart = (jCol / 3) \* 3;

// check the validity within boxes

**for** (**int** newRow = rowStart; newRow < rowStart + 3; newRow++) {

**for** (**int** newCol = colStart; newCol < colStart + 3; newCol++) {

**if** (board[newRow][newCol] == num) {

count2++;

}

**if** (count2 > 1) {

valid = **false**;

}

}

}

}

}

}

**return** valid;

}

// Is this a solved sudoku?

**public** **boolean** isComplete() {

**boolean** complete = **true**;

// while valid

**if** (isValid()) {

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

// if found any zeros its not complete

**if** (board[iRow][jCol] == 0) {

complete = **false**;

}

}

}

}

**return** complete;

}

// return true if val appears in the row of the puzzle

**private** **boolean** doesRowContain(**int** iRow, **int** val) {

**boolean** rowContains = **false**;

**for** (**int** jCol = 0; jCol < 9; jCol++) {

**if** (board[iRow ][jCol] == val) {

rowContains = **true**;

}

}

**return** rowContains;

}

// return true if val appears in the col (column) of the puzzle

**private** **boolean** doesColContain(**int** jCol, **int** val) {

**boolean** colContains = **false**;

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**if** (board[iRow][jCol ] == val)

colContains = **true**;

}

**return** colContains;

}

// return true if val appears in the 3 x 3 box

**private** **boolean** doesBoxContain(**int** row, **int** col, **int** val) {

**boolean** boxContain = **false**;

// find top left spot of the box

**int** rowStart = ((row ) / 3) \* 3;

**int** colStart = ((col ) / 3) \* 3;

**for** (**int** newRow = rowStart; newRow < rowStart + 3; newRow++) {

**for** (**int** newCol = colStart; newCol < colStart + 3; newCol++) {

**if** (board[newRow][newCol] == val) {

boxContain = **true**;

}

}

}

**return** boxContain;

}

// return n if n is the only possible value for this spot

// return 0 otherwise

**private** **int** fillSpot(Spot sq) {

**int** possibleVal = 0;

**int** num = 1;

**int** count = 0;

// go through numbers 1-9 and while the spot is not 0 try to fill it

**while** (num <= 9 && board[sq.getRow() ][sq.getCol() ] == 0) {

// if the number is not found in row,col,box

**if** (!doesRowContain(sq.getRow(), num)

&& !doesColContain(sq.getCol(), num)

&& !doesBoxContain(sq.getRow(), sq.getCol(), num)) {

possibleVal = num;

// increment count in case more than one occurance

count++;

}

num++;

}

// if same number has been found for more than one place return 0

**if** (count > 1) {

possibleVal = 0;

}

**return** possibleVal;

}

// return a valid spot if only one possibility for val in row

// return null otherwise

**private** Spot rowFill(**int** row, **int** val) {

Spot spot = **null**;

**int** count = 0;

**int** newCol = 0;

**for** (**int** jCol = 0; jCol < 9; jCol++) {

**if** (board[row ][jCol ] == 0) {

**if** (!doesRowContain(row, val)

&& !doesColContain(jCol, val)

&& !doesBoxContain(row, jCol, val)) {

newCol = jCol;

count++;

}

}

}

**if** (count == 1) {

spot = **new** Spot(row, newCol);

}

**return** spot;

}

// return a valid spot if only one possibility for val in col

// return null otherwise

**private** Spot colFill(**int** col, **int** val) {

Spot spot = **null**;

**int** count = 0;

**int** newRow = 0;

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**if** (board[iRow ][col ] == 0) {

**if** (!doesRowContain(iRow, val)

&& !doesColContain(col, val)

&& !doesBoxContain(iRow, col, val)) {

newRow = iRow;

count++;

}

}

}

**if** (count == 1) {

spot = **new** Spot(newRow, col);

}

**return** spot;

}

// return a valid spot if only one possibility for val in the box

// return null otherwise

**private** Spot boxFill(**int** rowbox, **int** colbox, **int** val) {

Spot spot = **null**;

**int** col = 0;

**int** row = 0;

**int** count = 0;

**int** nRow = ((rowbox - 1) / 3) \* 3;

**int** nCol = ((colbox - 1) / 3) \* 3;

**for** (**int** iRow = nRow; iRow < nRow + 3; iRow++) {

**for** (**int** jCol = nCol; jCol < nCol + 3; jCol++) {

**if** (board[iRow][jCol] != val) {

**if** (!doesRowContain(iRow , val)

&& !doesBoxContain(iRow , jCol , val)

&& !doesColContain(jCol , val) &&

board[iRow][jCol] == 0) {

row = iRow ;

col = jCol ;

count++;

}

}

}

}

**if** (count == 1) {

spot = **new** Spot(row, col);

}

**return** spot;

}

// solve sudoku!

**public** **void** solve() {

// makes a copy of the board.

**int**[][] tempBoard = **new** **int**[9][9];

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol<9; jCol++){

tempBoard[iRow][jCol]= board[iRow][jCol];

}

}

**int** count = 0;

// do it 3 times for assurance

**while** (count < 3) {

//fill spots that are 0 if possible

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

**if** (tempBoard[iRow][jCol] == 0) {

Spot spot = **new** Spot(iRow , jCol );

**int** num = fillSpot(spot);

**if** (num != 0) {

board[iRow][jCol] = num;

}

}

}

}

// use rowfill, colfill and boxfill now to fill the rest of it

**for** (**int** iRow = 0; iRow < 9; iRow++) {

**for** (**int** jCol = 0; jCol < 9; jCol++) {

**if** (tempBoard[iRow][jCol] == 0) {

// create a spot object to work with

Spot tempSpot = **new** Spot(iRow , jCol );

**int** num = 1;

**while** (num <= 9) {

tempSpot = rowFill(iRow , num);

**if** (tempSpot != **null**) {

board[tempSpot.getRow() ][tempSpot.getCol() ] = num;

}

tempSpot = colFill(jCol , num);

**if** (tempSpot != **null**) {

board[tempSpot.getRow() ][tempSpot.getCol()] = num;

}

tempSpot = colFill(iRow , num);

**if** (tempSpot != **null**) {

board[tempSpot.getRow() ][tempSpot.getCol() ] = num;

}

tempSpot = rowFill(jCol , num);

**if** (tempSpot != **null**) {

board[tempSpot.getRow() ][tempSpot.getCol() ] = num;

}

tempSpot = boxFill(iRow , jCol , num);

**if** (tempSpot != **null**) {

board[tempSpot.getRow() ][tempSpot.getCol() ] = num;

}

num++;

}

}

}

}

count++;

}

}

// who are you? Put your name here!

**public** **static** String myName() {

**return** "Ali Mojarrad";

}

}

**OutPut:**

Author: Ali Mojarrad

#0: Invalid board. Answers match.

#1: Invalid board. Answers match.

#2: Invalid board. Answers match.

#3: Invalid board. Answers match.

#4: Invalid board. Answers match.

#5: Invalid board. Answers match.

#6: Invalid board. Answers match.

#7: Invalid board. Answers match.

#8: Invalid board. Answers match.

#9: Solution found. Answers match. Solution found. Answers match.

#10: Solution found. Answers match. Solution found. Answers match.

#11: Solution found. Answers match. Solution found. Answers match.

#12: Not done yet. Answers match. Not done yet. Answers match.

#13: Not done yet. Answers match. Not done yet. Answers match.

#14: Solution found. Answers match. Solution found. Answers match.

#15: Not done yet. Answers match. Not done yet. Answers match.

#16: Not done yet. Answers match. Not done yet. Answers match.

#17: Not done yet. Answers match. Not done yet. Answers match.

#18: Not done yet. Answers match. Not done yet. Answers match.

#19: Solution found. Answers match. Solution found. Answers match.

#20: Not done yet. Answers match. Not done yet. Answers match.

#21: Solution found. Answers match. Solution found. Answers match.

#22: Solution found. Answers match. Solution found. Answers match.

#23: Solution found. Answers match. Solution found. Answers match.

#24: Solution found. Answers match. Solution found. Answers match.

#25: Not done yet. Answers match. Not done yet. Answers match.

#26: Solution found. Answers match. Solution found. Answers match.

Constructor check:

Answers match.

Answers match.

Answers match.

Answers match.

Author: Ali Mojarrad

**What I did to Fix it ?**

In the original program I forgot to include the box checking in my fillspot

And also boxfill method in my solve method. I also found out repeating solve method using count for a few times makes a lot of different which I did as well.