

Assignment Eleven – Java Collection TreeSet, 2D Arrays, and Exceptions

Problem Description

In this eleventh assignment, you will be using the Java Collection TreeSet class to solve a problem that is particularly well suited for mathematical sets. You will be using a 2D array to represent data that is in a matrix form. You will also be using exceptions to create a robust program that reduces errors and provides meaningful messages about what problems occur.

Learning Goals

Learn about using binary search trees to solve interesting problems, storing data in 2D arrays, and using exception handling to provide a robust program.

Problem Scenario

Over the weekend you had a surprise visit from Mr. Bubbles! He stopped by because he and Big Mommy were delighted with your programming skills, as now the neighborhood is once again safe and there are enough cookies for everyone. He even brought you a container of freshly baked chocolate chip and melange cookies (whatever that might be). It turns out he's a pretty nice guy and got the name "Mr. Bubbles" because he enjoys fizzy drinks. So, you kick back with him enjoying a nice cold beer, a root beer.

As you are nibbling surprisingly tasty cookies and enjoying fizzy bliss, Mr. Bubbles tells you that his brother Howard has a problem that he thinks you can help with. Howard really likes solving Sudoku puzzles. He is often unsure if he has successfully solved a puzzle and sometimes he gets stuck as well and really needs a hint. Mr. Bubbles wants you to help his brother out of his puzzling predicament with your near legendary programming preeminence!

Obtaining and Unzipping the NetBeans Assignment Template Project Files

As usual, a NetBeans template project has been provided for you to complete this assignment.

Renaming Your Project Name and Project Folder

After downloading and unzipping the template, you need to rename the project name and project folder as you have done in the previous assignments.

Using the Assignment Template Project Files

It is important to know exactly what the provided template files give you and what you need to actually complete for this assignment.

Provided For You:

- A project that is structurally complete, including the input and output directories
- Fully functional and documented CS310Assignment11 class (Do not modify)
- Minimally functional SudokuBoard class
- Minimally functional SudokuSolutionHelper class

To Complete:

- SudokuBoard class – method code and documentation
- SudokuSolutionHelper class – method code and documentation

Each of the areas in the classes or methods that need to be completed has a comment inside the method body similar to this:

```
// TODO - complete this method
```

Once you complete a class or method, remove these comments (i.e., the TODO comments).

DO NOT CHANGE the method signatures, data member names (attributes), constant names, or access specifiers (public / private / static).

You are also responsible for completing any of the missing required Javadoc comments that are part of the course coding standards. You can use the Javadoc commenting in other classes of the project as an example for writing your own comments.

Note: if there is already an existing author for a file that you modify, do not remove it. Instead you will be adding your own author to the file. If you do not modify a file, do not add your author information to it.

Assignment Detailed Specifications:

In this assignment you will be reading in Sudoku puzzles stored in text files and storing them in a SudokuBoard class containing a two dimensional array. Puzzles can be any size from 4x4, 9x9, 16x16, to 25x25 cells. These will simply be referred to as puzzles of board size 4, 9, 16, and 25. You will also be making use of sets to help determine the state of the Sudoku board, which will provide the functionality for the SudokuSolutionHelper class.

If you are unfamiliar with Sudoku, this Wikipedia entry has a lot of information:

<https://en.wikipedia.org/wiki/Sudoku>

Reading and Representing Sudoku Puzzles

Individual Sudoku puzzles are stored in plain text files. The first line of each file specifies the "size" of the game. This number will always be a perfect square, such as 4 or 9. Following that line is a sequence of lines that contain the values for each "cell" of the board. For a Sudoku board of size N, the values in the cells can be 1 to N inclusive. In the input specification, we use the zero value to represent a "blank" cell. For example, the following input file (puzzleExample.txt):

```
4
0 4 1 0
1 3 0 0
0 2 0 0
0 0 3 2
```

Represents the following Sudoku board:

puzzleExample.txt Layout	4 Mini-Grids																																
<table><tr><td></td><td>4</td><td>1</td><td></td></tr><tr><td>1</td><td>3</td><td></td><td></td></tr><tr><td></td><td>2</td><td></td><td></td></tr><tr><td></td><td></td><td>3</td><td>2</td></tr></table>		4	1		1	3				2					3	2	<table><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>2</td><td>2</td><td>3</td><td>3</td></tr><tr><td>2</td><td>2</td><td>3</td><td>3</td></tr></table>	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	4	1																															
1	3																																
	2																																
		3	2																														
0	0	1	1																														
0	0	1	1																														
2	2	3	3																														
2	2	3	3																														

In the rules of Sudoku, a cell cannot contain the same value of another cell in the same row, column, or mini-grid. For a board of size N, the mini-grid that contains a cell is the $\sqrt{N} * \sqrt{N}$ region that contains that cell. For example, the mini-grid containing (0, 0) is darkened in the preceding diagram. In this example board, there are four rows [0, 3], four columns [0, 3], and four mini-grids [0, 3].

SudokuBoard Class Specifications

SudokuBoard UML Diagram:

SudokuBoard
- board: Integer[][] - boardSize: int
+ SudokuBoard(String filename) + getBoardSize(): int + getCell(int row, int col): Integer + setCell(int row, int col, Integer value): void + toString(): String

Data Members:

The UML diagram for the SudokuBoard class indicates that there are two data members in this class. These have been provided for you in the template.

private Integer[][] board;

This is the 2D array used for storing the Sudoku board cell values. In a 2D array, the first index is the row and the second index is the column that you wish to access.

private int boardSize;

This is the size of the Sudoku board, which will also sometimes be referred to as the value N. The Sudoku board size represents the size of a full row or column of the board and the entire board contains N x N cells. The board size is also a perfect square number.

Constructor:

The SudokuBoard class has only one constructor.

public SudokuBoard(String filename) throws FileNotFoundException

The constructor attempts to open the specified file and then attempts to read in the values for the puzzle and put them into the 2D array. This constructor will throw various exceptions with custom messages for issues that it encounters. This is the general algorithm for the constructor:

```
Create a new File object with the filename passed in
Try creating a Scanner object to read from the file
Catch the Scanner creation failing,
    Throw a new FileNotFoundException with message:
    "SudokuBoard: File input/foobar.txt was not found or could
    not be opened!"
Try reading the first line of the file and integer parse it
Catch the parsing failing
    Throw a new NumberFormatException with message:
    "SudokuBoard: Board size AA is not an integer!"
If the board size is not a perfect square
    Throw a new IllegalArgumentException with message:
    "SudokuBoard: Board size 10 is not a perfect square number!"
Initialize the 2D array with the correct board size dimensions
Initialize row count to be zero
Initialize the line count to be one
Loop while the file has more lines to read
    Increment the line count
    Get the next line in the file
    Split the line based on a space delimiter character
    If the split line length is greater than the board size
        Throw a new IllegalArgumentException with message:
        "SudokuBoard: Line 1 length of 10 is incorrect!"
    Loop over the split line array
        Try integer parsing the split line value
        Catch the parsing failing
            Throw a new NumberFormatException with message:
            "SudokuBoard: Board cell (0, 0) value of XYZ is not
            an integer!"
        Try setCell method with the row, column, and value
        Catch IndexOutOfBoundsException if row/column invalid
        If the file line count is not correct
```

```

        Throw new IllegalArgumentException with message:
        "SudokuBoard: 10 lines in file is incorrect!"
    Increment the row count
    If the file line count is not correct
        Throw new IllegalArgumentException with message:
        "SudokuBoard: 10 lines in file is incorrect!"
    Close the Scanner object

```

Important Note: A try/catch block will not trigger some of the exceptions you throw. Some will be triggered by conditional testing situations. This will happen in other methods that you are writing as well. If you see “throw new” it means you are doing something like this in your code:

```

if (conditional-test-expression) {
    throw new ExceptionName ("Error Message");
}

```

Important Note: Some of the values in the exception messages are merely placeholders, such as “foobar.txt”, 10, and (0, 0). These will be replaced by the actual values passed to the method, from the file itself, or for the row/column that is being set. The values have been bolded and italicized to make it a little easier to see in the algorithm.

Getters & Setters:

The SudokuBoard class has two getters methods and one setter method.

```
public int getBoardSize()
```

This method returns the size of the Sudoku board puzzle. Remember, this is merely N in the puzzle and not the number of total cells in the puzzle.

```
public Integer getCell(int row, int col)
```

This method returns the integer cell value at the row and column location in the 2D array. If the row is invalid or the column is invalid, a new IndexOutOfBoundsException will be thrown with these messages respective to what is incorrect:

```
"SudokuBoard.getCell: Row " + row + " is out of bounds!"
```

```
"SudokuBoard.getCell: Column " + col + " is out of bounds!"
```

The displayed row/column value could also be a positive or negative value.

```
public void setCell(int row, int col, Integer value)
```

This method sets the integer cell value at the row and column location in the 2D array. If the row is invalid or the column is invalid, a new IndexOutOfBoundsException will be thrown with these messages respective to what is incorrect:

```
"SudokuBoard.setCell: Row " + row + " is out of bounds!"
```

```
"SudokuBoard.setCell: Column " + col + " is out of bounds!"
```

The displayed row/column value could also be a positive or negative value.

If the integer value passed to the method is null, a new `IllegalArgumentException` is thrown with this message:

```
"SudokuBoard.setCell: Cell value is null!"
```

SudokuBoard Methods:

The `SudokuBoard` class has one additional overridden method that needs to be implemented.

@Override

```
public String toString()
```

This method creates a string representation of the `SudokuBoard` object. Using the example from earlier in the description, the `toString` method would generate this string representation:

	0	1	2	3
0	0	4	1	0
1	1	3	0	0
2	0	2	0	0
3	0	0	3	2

It is recommended that you use the `StringBuilder` and `String` formatting to generate this table. Here is a mini-example of using both to create a string containing the values 0 to 9 each on a separate line and then printing out the `toString` value of the `StringBuilder` object.

```
StringBuilder strBuilder = new StringBuilder();  
String strFormat;
```

```
for(int i = 0; i < 10; i++)  
{  
    strFormat = String.format("%d\n", i);  
    strBuilder.append(strFormat);  
}
```

```
System.out.println(strBuilder.toString());
```

A properly aligned and formatted table should work for valid board sizes of 4, 9, 16, and 25. See the provided output files for additional examples of how things will look.

<https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html>

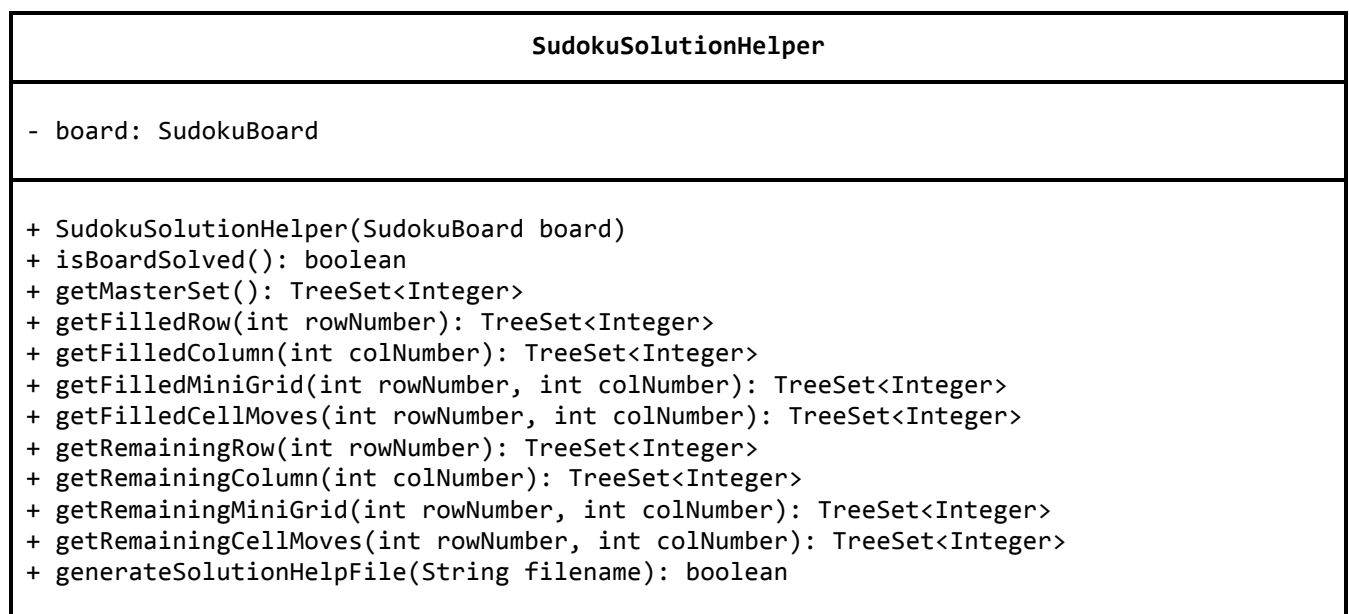
SudokuSolutionHelper Class Specifications

The SudokuSolutionHelper class will make use of the Java Collection TreeSet, which is a balanced binary search tree. You will be using specific operations of the TreeSet to accomplish the tasks in this class. More information about the TreeSet can be found here:

<https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html>

The specific TreeSet methods that you will need are these: add, addAll, removeAll, and toString. The addAll method is used for set union, while the removeAll method is used for set intersection.

SudokuSolutionHelper UML Diagram:



Data Members:

The UML diagram for the SudokuSolutionHelper class indicates that there is one data member in this class. It has been provided for you in the template.

private SudokuBoard board;

This is the SudokuBoard object that the solution helper will work with.

Constructor:

The SudokuSolutionHelper class has only one constructor.

```
public SudokuSolutionHelper(SudokuBoard board)
```

The constructor attempts to set the SudokuBoard reference to the object passed into the constructor. If the reference passed in is null, a new IllegalArgumentException will be thrown with this message:

```
"SudokuSolutionHelper: Board parameter is null!"
```

SudokuSolutionHelper Methods:

The SudokuSolutionHelper class has eleven additional methods that need to be completed. These are mostly similar to getter methods, although they are not getting class data members directly. The solution helper will be making use of sets to help determine various things about the Sudoku board state.

```
public TreeSet<Integer> getMasterSet()
```

This method creates and returns a new TreeSet that contains the values 1 to N inclusive. This corresponds to the possible values that can occur in a row, column, and mini-grid in a Sudoku puzzle. If the board size is four, then the returned TreeSet will contain the values 1, 2, 3, and 4. Use a loop to add the correct values to the set.

```
public TreeSet<Integer> getFilledRow(int rowNumber)
```

This method returns a set of the non-zero cell values for the given row. The Sudoku board contains N rows in the range of 0 to (N – 1). If the row is invalid, a new IndexOutOfBoundsException is thrown with the following message:

```
"SudokuSolutionHelper.getFilledRow: " +  
"Invalid row number of " + rowNumber + "!"
```

Row zero is darkened in the example below. The filled set would only contain the values 1 and 4 in this case as the other two cells in the row contain zeros:

	4	1	
1	3		
	2		
		3	2


```
public TreeSet<Integer> getFilledColumn(int colNumber)
```

This method returns a set of the non-zero cell values for the given column. The Sudoku board contains N columns in the range of 0 to (N – 1). If the column is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getFilledColumn: " +  
"Invalid column number of " + colNumber + "!"
```

Column zero is darkened in the example below. The filled set would only contain the value 1 in this case as the other three cells in the column contain zeros:

	4	1	
1	3		
	2		
		3	2

```
public TreeSet<Integer> getFilledMiniGrid(int rowNumber, int  
colNumber)
```

This method returns a set of the contents for the given mini-grid that contains the (row, col) cell. The Sudoku board contains N mini-grids, each $\sqrt{N} * \sqrt{N}$ in size. If the row is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getFilledMiniGrid: " +  
"Invalid row number of " + rowNumber + "!"
```

If the column is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getFilledMiniGrid: " +  
"Invalid column number of " + colNumber + "!"
```

Mini-grid (0, 0) is darkened in the example below. The filled set would contain the values 1, 3, and 4 in this case as the other cell in the mini-grid contains a zero. The cells (0, 0), (0, 1), (1, 0), and (1, 1) would all result in the same set being returned by this method.

Hint: You will need two loops to accomplish this goal and also use some math to obtain the upper left cell of the mini-grid. The square root of the board size will be helpful along with some simple math operations. Drawing pictures may help.

	4	1	
1	3		
	2		
		3	2

```
public TreeSet<Integer> getFilledCellMoves(int rowNumber, int colNumber)
```

This method returns the set union of the filled row, filled column, and filled mini-grid. This set contains the filled moves at a particular cell location in the Sudoku board. If the row is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getFilledCellMoves: " +
"Invalid row number of " + rowNumber + "!"
```

If the column is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getFilledCellMoves: " +
"Invalid column number of " + colNumber + "!"
```

Important Note: You are performing the set union operation here on two sets. Normally, you would want the set operations to generate and return a new set. In Java, this is not what happens. The “this” set is actually modified during the set operations of union, intersection, and difference (the “this” set is the object being acted upon by the method call, not the parameter passed in).

```
public TreeSet<Integer> getRemainingRow(int rowNumber)
```

This method returns the set containing any moves remaining in a specific row by using set difference with the master set and filled row set. If the row is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getRemainingRow: " +
"Invalid row number of " + rowNumber + "!"
```

Important Note: The ordering of the two sets in set difference matters!

```
public TreeSet<Integer> getRemainingColumn(int colNumber)
```

This method returns the set containing any moves remaining in a specific column by using set difference with the master set and filled column set. If the column is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getRemainingColumn: " +  
"Invalid column number of " + colNumber + "!"
```

```
public TreeSet<Integer> getRemainingMiniGrid(int rowNumber, int  
colNumber)
```

This method returns the set containing any moves remaining in a specific mini-grid by using set difference between the master set and filled mini-grid set. If the row is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getRemainingMiniGrid: " +  
"Invalid row number of " + rowNumber + "!"
```

If the column is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getRemainingMiniGrid: " +  
"Invalid column number of " + colNumber + "!"
```

```
public TreeSet<Integer> getRemainingCellMoves(int rowNumber, int  
colNumber)
```

This method returns the difference of the filled cell moves and the master set. This set contains the remaining moves at a particular cell location in the Sudoku board. If the row is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getRemainingCellMoves: " +  
"Invalid row number of " + rowNumber + "!"
```

If the column is invalid, a new `IndexOutOfBoundsException` is thrown with the following message:

```
"SudokuSolutionHelper.getRemainingCellMoves: " +  
"Invalid column number of " + colNumber + "!"
```

```
public boolean isBoardSolved()
```

This method determines whether the Sudoku board has been solved. It returns true if the board has been solved and false otherwise. If any cell on the board has remaining moves, the board is not yet solved.

```
public boolean generateSolutionHelpFile(String filename)
```

This method generates a help file that can be used to see what is remaining for solving the Sudoku puzzle. Help is provided for each row, column, mini-grid, and cell. The solution help will be written to files in the “output” directory with the same filename as the original puzzle, except that “help-” will be appended to the filename. This is done automatically for you. This is

the general algorithm, where everything is being printed/written to the file except for the PrintWriter error messages that are output to standard output:

```
Create a new File object with the filename passed in
Try creating a PrintWriter object to write to the file
Catch PrintWriter creation failing FileNotFoundException (fnfe)
    Output "SudokuSolutionHelper.generateSolutionHelpFile: "
    Output "ERROR: File could not be created! " + fnfe
Return false
```

```
Print heading: "4 x 4 Sudoku Puzzle Values"
Print dashed line
Print the toString of the SudokuBoard object
```

```
If the board is solved
    Print "Congratulations! This Sudoku puzzle has been solved!"
Else
    Print "-- Row Help --"
    Loop through the rows in the Sudoku board
        Get the remaining moves at the specific row
        If the remaining moves set is not empty
            Print "Row 1 Remaining Moves: [1, 3]\n"
    Print a newline character
    Print "-- Column Help --"
    Loop through the columns in the Sudoku board
        Get the remaining moves at the specific column
        If the remaining moves set is not empty
            Print "Column 1 Remaining Moves: [1, 3]\n"
    Print a newline character
    Print "-- Mini-Grid Help --"
    Loop through the mini-grid rows in the Sudoku board
        Loop through the mini-grid columns in the Sudoku board
            Get the remaining moves at the specific mini-grid
            If the remaining moves set is not empty
                Print "Mini-Grid (0, 0) Remaining Moves: [1, 3]\n"
    Print a newline character
    Print "-- Cell Help --"
    Loop through the rows in the Sudoku board
        Loop through the columns in the Sudoku board
            Get the remaining moves at the specific cell
            If the remaining moves set is not empty
                Print "Cell (0, 0) Remaining Moves: [2, 3]\n"
    Print a newline character
Flush the PrintWriter object
Close the PrintWriter object
```

Note: Specific integer literals used in the algorithm are merely placeholders and will be replaced by specific values from the puzzles loaded.

Testing Using the CS310Assignment11 Class

To know if your SudokuBoard and SudokuSolutionHelper classes are working, you can partially test them by using the “Test Sudoku Board and Solution Helper” option and visually verifying the results against the provided output file. This does not exercise everything in the code, so there still might be cases that you will have to test.

To test everything more fully, you will have to try different Sudoku puzzles, both incomplete and solved. Some puzzle text files are provided in the input directory that you can use. They do not test all scenarios. There are also several invalid boards to use for testing your exceptions.

CS310 Class Playground Method

Once again, you are provided with a mostly empty “playground” method in the main class so that you can exercise your code. You can test however you want to in this method. The code still needs to compile, follow the coding standards, and not cause the program to crash.

Program Output

See the separate Assignment 11 Program Output files for program output.

Commenting

All methods, including those overridden, must have Javadoc comments. All classes need Javadoc comments, including a description of the class, the author, and the version. In-method commenting is also required to explain complicated logic.

Expert’s Corner - What Works and What Doesn’t

Dealing with potential program errors is an important task and can take a lot of work. It can be difficult to see all of the potential problems that might occur as well. Sometimes it is possible to rely on method return values to help address errors. This may not be robust enough, as only one value can be returned and we might need the returned value for a different purpose. Exceptions are a way to help address these issues, as there is a catch-all type exception for unexpected problem cases and multiple different exceptions can be generated from the same code blocks.

Having a lot of try/catch blocks can also make the program logic harder to read and understand, so there is definitely a trade off. Being able to throw exceptions with custom messages is a very powerful tool, as it allows for better descriptions about what has gone wrong. In general, it is best not to rely solely on the catch-all exception case, as it makes it non-specific and makes it more difficult to address and correct problems.

I think that we can all agree that learning to use and handle exceptions is an important skill on the road to becoming an *exceptional* programmer! (Pun intended?!?)

Sudoku and sets go together perfectly. The TreeSet is a great tool here, as it guarantees that the sets will be displayed how we normally display mathematical sets of numbers (ascending sorted). The HashSet does not make this guarantee, so displaying the sets might not look as pleasing. For

instance, if the numbers 1, 15, and 25 are added both to a TreeSet and a HashSet and then the results are displayed, you would get output similar to this:

```
Tree Set: [1, 15, 25]
Hash Set: [1, 25, 15]
```

The reason that that TreeSet displays the results in ascending sorted order is because it is a special type of data structure called a binary search tree. It is always possible to obtain a sorted ordering of the data in the tree in $O(n)$ time. If you remember your previous work with hash tables, the data is not ordered in any meaningful way from a user perspective.

You also indirectly were using recursion during this assignment, as the TreeSet uses it for many of the tree operations. You will be exploring recursion and trees more fully in the next assignment.

This assignment is also a great starting point for making a full-featured Sudoku game or making a Sudoku solving program.

Program Submission

This programming assignment is due by the time indicated in the online course calendar.

You will submit a zip file containing your project files. To submit your project:

- First export your project from NetBeans. To do this:
 - Highlight the project name.
 - Click on **File** from the top menu, and select **Export Project**.
 - Select **To ZIP**
 - Name your export file in the following format:
CS310Assignment11<firstname><lastname>.zip

For example:

CS310Assignment11JohnSmith.zip

NOTE: Save this zip file to some other directory, not your project directory.

- Then submit your **.zip** file to the **Prog Assn 11** Submission Folder (located under **Assignments** tab in the online course).

Warning: Only NetBeans export files will be accepted.

Do not use any other kind of archive or zip utility.

Warning: Projects that do not compile will not be accepted and will receive a grade of zero.

Warning: Reading assignment descriptions may be hazardous to your health.

Grading

This program will be graded using the **rubric** that is linked under the **Assignments** tab in the online course. Click on the specific assignment to see the correct rubric.

Late Policy

See your instructor's syllabus for details regarding their specific late policy.