**Advanced Java**

**Chapter 8 (Multi-Dimensional Arrays)**

**Declaring, initializing, and displaying the contents of two-dimensional arrays:**

…sample code…

**Review of File I/O**

…sample code…note that Scanner is designed for whitespace-delimited streams…

Modify this code to read a file of 81 ints into a 9x9 array and display that array.

**Backtracking and Recursion**

Pretty much *any* numeric puzzle can be solved with three Boolean methods:

isCompletePuzzle()

// returns true if every spot in the puzzle has a value, false otherwise

isValidPuzzle()

// returns true if the puzzle contains no counterexamples, false otherwise

isSolvedPuzzle()

// returns true if the puzzle is both valid and complete, false otherwise

The solution strategy is implemented in a fourth method

solvePuzzle()

which executes the following recursive algorithm:

// base cases

Is the puzzle valid? If not, return false.

Is the puzzle complete? If so, return true (valid and complete!)

// current candidate is valid and incomplete, so we have a recursive case

Locate the first blank space in the puzzle.

Drop in a candidate.

Make a recursive call to solvePuzzle() with the updated puzzle.

Got back a true? Return true.

Got back a false? Try the next candidate and make another recursive call.

No next candidate? Reset the candidate cell to blank and return false.

**HOMEWORK: SUDOKU SOLVER**

Write a Java program that performs the following tasks:

* Display a friendly greeting to the user
* Prompt the user for a filename (if the file wasn’t provided on the command line)
* Accept that filename
* Attempt to open the file
* Read the 81 numbers in the file and populate a 9x9 table with them
  + Note: the char \* or - is used to designate an empty spot

because 0 is a valid entry in the hexoduko puzzle

* Display the contents of the file
* Solve the puzzle using the algorithm given
* Display the solved puzzle, or an appropriate message if no solution is found

The rules of Sudoku:

The integers 1-9 appear in a 9x9 grid. The grid is subdivided into nine 3x3 subgrids.

The thee Sudoku constraints are:

Each digit 1-9 appears in each row, without omissions or duplicates

Each digit 1-9 appears in each column, without omissions or duplicates

Each digit 1-9 appears in each 3x3 subgrid, without omissions or duplicates

“Diagonal Rule” Sudoku adds a fourth constraint:

Each digit 1-9 appears in each of the two long diagonals, without omissions or duplicates.

The “isValidPuzzle” method need check only these constraints; if they are satisfied, we’ll call it a “valid” puzzle.

The input file will be 81 integers, space-delimited. Test your program with a variety of Sudoku puzzles. They are easily found on the ‘net. How long does it take your program to solve the World’s Hardest Sudoku? Or an empty puzzle?

Note that if you get the regular version working, the Super version only differs by a few lines in the isValidPuzzle method. However, be careful how you test it; most regular Sudoku puzzles aren’t diagonal-rule Sudokus and you’ll need to find specific test cases for them.

You can solve other number puzzles (like Numbrix) with this technique by dropping in the appropriate validator. The rest of the code is unchanged.

The standard solver is worth full credit; a diagonal-rule solver gets you an additional 20 points. Throw in a Numbrix solver for yet another 20 points. For yet another twenty points there is the 16x16 Hexudoku, with sixteen 4x4 regions and entries in hexadecimal (0-9 and A-F).

Note: if you are thinking ahead you can write an entirely generic Sudoku solver that will handle 9x9, 16x16, 25x25, and 36x36 puzzles (using 0-9 and A-Z gets you to base 36). The limitations are that you run out of symbols to represent the values if you go beyond base 36, and the algorithm is exponential so the solve times on larger puzzles can become…large.