**Supplement: Case Study: Sudoku**

**For Introduction to Java Programming**

**By Y. Daniel Liang**

This case study can be presented after Chapter 7, “Multidimensional Arrays.”

This book teaches you how to program using a wide variety of problems with various levels of difficulty. We use simple, short, and stimulating examples to introduce programming and problem-solving techniques and use interesting and challenging examples to motivate students in programming. This supplement presents an interesting problem of a sort that appears in the newspaper every day. It is a number-placement puzzle, commonly known as *Sudoku*.

## 1 Problem Description

***<Side Remark: fixed cells>***
***<Side Remark: free cells>***

Sudoku is a 9×9 grid divided into smaller 3×3 boxes (also called regions or blocks), as shown in Figure 1(a). Some cells, called *fixed cells*, are populated with numbers from 1 to 9. The objective is to fill the empty cells, also called *free cells*, with numbers 1 to 9 so that every row, every column, and every 3×3 box contains the numbers 1 to 9, as shown in Figure 1(b).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | | | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

(a) Input

Solution →

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

(b) Output

**Figure 1**
*(b) is the solution to the Sudoku puzzle in (a).*

***<Side Remark: representing a grid>***

For convenience, we use value 0 to indicate a free cell, as shown in Figure 2(a). The grid can be naturally represented using a two-dimensional array, as shown in Figure 2(a).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 0 | 0 | 7 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 9 | 5 | 0 | 0 | 0 |
| 0 | 9 | 8 | 0 | 0 | 0 | 0 | 6 | 0 |
| 8 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 3 |
| 4 | 0 | 0 | 8 | 0 | 3 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 6 |
| 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 4 | 1 | 9 | 0 | 0 | 5 |
| 0 | 0 | 0 | 0 | 8 | 0 | 0 | 7 | 9 |

(a)

```
int[][] grid =
  {{5, 3, 0, 0, 7, 0, 0, 0, 0},
   {6, 0, 0, 1, 9, 5, 0, 0, 0},
   {0, 9, 8, 0, 0, 0, 0, 6, 0},
   {8, 0, 0, 0, 6, 0, 0, 0, 3},
   {4, 0, 0, 8, 0, 3, 0, 0, 1},
   {7, 0, 0, 0, 2, 0, 0, 0, 6},
   {0, 6, 0, 0, 0, 0, 2, 8, 0},
   {0, 0, 0, 4, 1, 9, 0, 0, 5},
   {0, 0, 0, 0, 8, 0, 0, 7, 9}
  };
```

(b)

**Figure 2**

*A grid can be represented using a two-dimensional array.*

*2 Problem-Solving Strategy*

How do you solve this problem? An intuitive approach to solve this problem is to employ the following three rules:

Rule 1: Fill in free cells from the first to the last.
Rule 2: Fill in a smallest number possible.
Rule 3: If no number can fill in a free cell, backtrack.

For example, you can fill 1 into grid[0][2], 2 into grid[0][3], 4 into grid[0][5], 8 into grid[0][6], and 9 into grid[0][7], as shown in Figure 3(a).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | *1* | *2* | 7 | *4* | *8* | *9* | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | | | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

(a)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | *1* | *2* | 7 | *4* | *9* | *8* | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | | | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

(b)

**Figure 3**

*The program attempts to fill in free cells.*

***<Side Remark: backtrack>***

Now look at grid[0][8]. There is no possible value to fill in this cell. You need to backtrack to the previous free cell at grid[0][7] and reset its value. Since grid[0][7] is already 9, no new value is possible. So you have to backtrack to its previous free cell at grid[0][6] and change its value to 9. Continue to move forward to set grid[0][7] to 8, as shown in Figure 3(b). Now there is still no possible value for grid[0][8]. Backtrack to grid[0][7], no possible new value for this cell. Backtrack to grid[0][6], no possible new value for this cell. Backtrack to grid[0][5] and change it to 6. Now continue to move forward.

The search moves forward and backward continuously until one of the following two cases arises:

- All free cells are filled. A solution is found.
- The search is backtracked to the first free cell with no new possible value. The puzzle has no solution.

Pedagogical NOTE

***<side remark: Sudoku animation>***

Follow the link www.cs.armstrong.edu/liang/animation/SudokuAnimation.html to see how the search progresses. As shown in Figure 4(a), number 1 is placed in the first row and last column. This number is invalid, so, the next value 2 is placed in Figure 4(b). This number is still invalid, so, the next value 3 is placed in Figure 4(c). The simulation displays all the search steps.

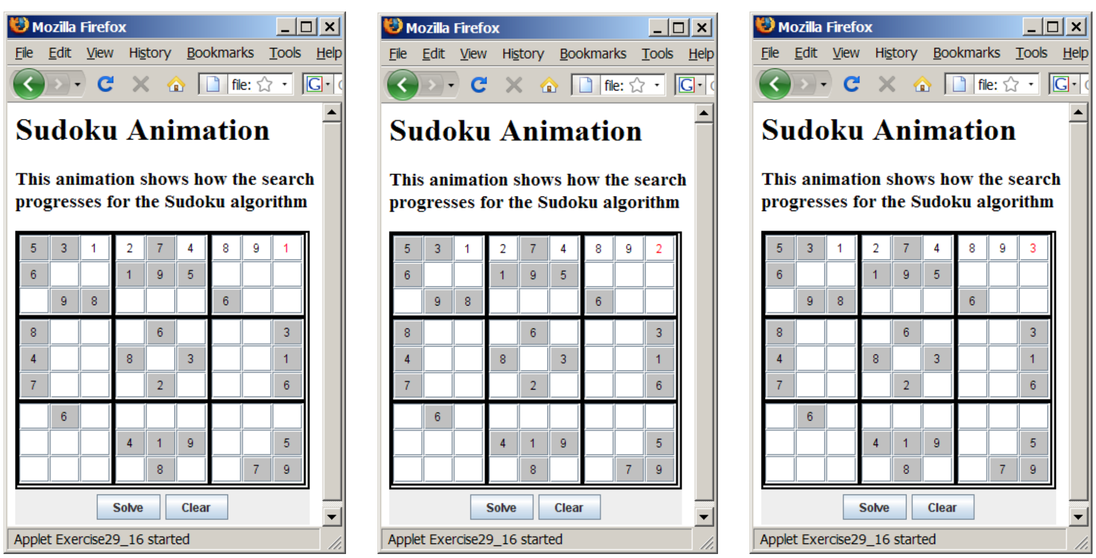



**Figure 4**

*The animation tool enables you to observe how the search works for solving a Sudoku puzzle.*

******End NOTE***

*3 Program Design*

The program can be designed as follows in (a) and further refined with methods in (b):

```
Read the input for a puzzle;
if (the grid is not valid)
  Report the grid not valid;
else {
  Search for a solution;
  if (solution found)
    Display the solution;
  else
    Report no solution;
}
```

(a)

Refined →

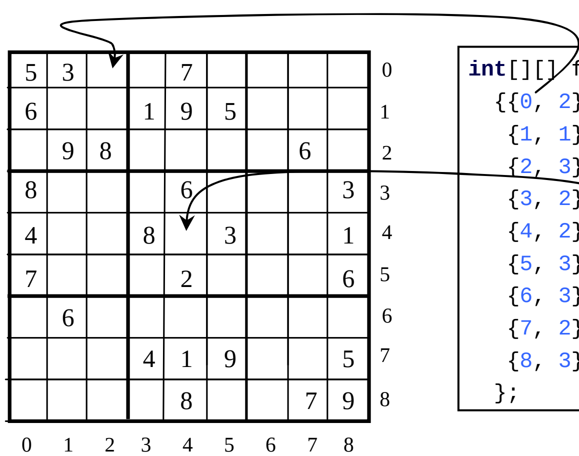
```
int[][] grid =
  readAPuzzle(grid);
if (!isValid(grid))
  Report the grid not valid;
else {
  if (search(grid))
    prindGrid(grid);
  else
    Report no solution;
}
```

(b)

The readAPuzzle method reads a Sudoku puzzle from the console into grid. The printGrid method displays the contents in grid to the console. The isValid method checks whether the grid is valid. These methods are easy to implement. We now turn our attention to the search method.

*4 Search Algorithm*

To better facilitate search on free cells, the program stores free cells in a two-dimensional array, as shown 8.a. Each row in the array has two columns, which indicate the subscripts of the free cell in the grid. For example, {freeCellList[0][0], freeCellList[0][1]} (i.e., {0, 2}) is the subscript for the first free cell grid[0][2] in the grid and {freeCellList[25][0], freeCellList[25][1]} (i.e., {4, 4}) is the subscript for free cell grid[4][4] in the grid, as shown in Figure 5.




```
int[][] freeCellList =
  {{0, 2}, {0, 3}, {0, 5}, {0, 6}, {0, 7}, {0, 8},
   {1, 1}, {1, 2}, {1, 6}, {1, 7}, {1, 8}, {2, 0},
   {2, 3}, {2, 4}, {2, 5}, {2, 6}, {2, 8}, {3, 1},
   {3, 2}, {3, 3}, {3, 5}, {3, 6}, {3, 7}, {4, 1},
   {4, 2}, {4, 4}, {4, 6}, {4, 7}, {5, 1}, {5, 2},
   {5, 3}, {5, 5}, {5, 6}, {5, 7}, {6, 0}, {6, 2},
   {6, 3}, {6, 4}, {6, 5}, {6, 8}, {7, 0}, {7, 1},
   {7, 2}, {7, 6}, {7, 7}, {8, 0}, {8, 1}, {8, 2},
   {8, 3}, {8, 5}, {8, 6}
  };
```

**Figure 5**

*freeCellList is a two-dimensional array representation for the free cells.*

The search starts from the first free cell with k = 0, where k is the index of the current free cell being considered in the free cell list, as shown in Figure 6. It fills a smallest possible valid value in the current free cell and then moves forward to consider the next cell. If no valid value can be found for the current free cell, the search backtracks to the preceding free cell. This process continues until all free cells are filled with valid value (a solution found) or search backtracks to the first free cell with no solution.

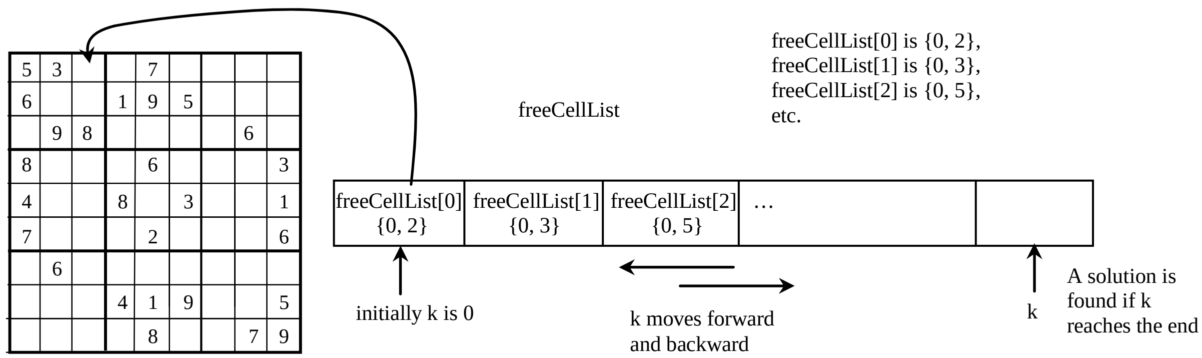



**Figure 6**

*The search attempts to fill appropriate values in free cells.*

The search algorithm can be described as follows:

Step 1: (Initialization) Obtain a freeCellList from a grid, as shown in Figure 5. Let k denote the index in freeCellList with k initially 0, as shown in Figure 6.

Repeatedly perform Steps 2-4 until search ends with a solution or no solution

{

Step 2: Let grid[i][j] be the current free cell being considered, where i = freeCellList[k][0] and j = freeCellList[k][1].

Step 3: If grid[i][j] is 0, fill it with 1.

Step 4: Consider three cases:

***<Side Remark line 89: solution found>***

Case 1: grid[i][j] is valid. If k is the last index in freeCellList, *a solution is found*. Otherwise, search moves forward with k = k + 1.

Case 2: grid[i][j] is invalid and grid[i][j] < 9. Set a new value for the free cell with grid[i][j] = grid[i][j] + 1.

***<Side Remark line 89: no solution>***

Case 3: grid[i][j] is invalid and grid[i][j] is 9. If k = 0, *search ends with no solution*. Otherwise backtracks with k = k – 1, reset i = freeCellList[k][0] and j = freeCellList[k][1], and continue to backtrack if grid[i][j] is 9. When grid[i][j] < 9, set grid[i][j] = grid[i][j] + 1.

}

*5 Implementation*

Listing 1 gives the source code for the program.

**Listing 1 Sudoku.java**

******PD: Please add line numbers in the following code******

***<Side Remark line 6: read input>***
***<Side Remark line 8: input valid?>***
***<Side Remark line 10: search>***
***<Side Remark line 12: print result>***
***<Side Remark line 19: read input>***
***<Side Remark line 29: return grid>***
***<Side Remark line 33: get free-cell list>***
***<Side Remark line 35: count free cells>***
***<Side Remark line 42: create free cell list>***
***<Side Remark line 55: print grid>***
***<Side Remark line 64: search a solution>***
***<Side Remark line 70: continuous search>***
***<Side Remark line 74: start with 1>***
***<Side Remark line 76: is valid?>***
***<Side Remark line 78: found>***
***<Side Remark line 81: to next free cell>***
***<Side Remark line 86: increase cell value>***
***<Side Remark line 91: no solution>***
***<Side Remark line 93: reset cell value>***
***<Side Remark line 94: backtrack>***
***<Side Remark line 107: check valid>***
***<Side Remark line 109: check row>***
***<Side Remark line 114: check column>***
***<Side Remark line 119: check box>***
***<Side Remark line 128: valid grid?>***

```
import java.util.Scanner;
```

```java
public class Sudoku {
  public static void main(String[] args) {
    // Read a Sudoku puzzle
    int[][] grid = readAPuzzle();

    if (!isValid(grid))
      System.out.println("Invalid input");
    else if (search(grid)) {
      System.out.println("The solution is found:");
      printGrid(grid);
    }
    else
      System.out.println("No solution");
  }

  /** Read a Sudoku puzzle from the keyboard */
  public static int[][] readAPuzzle() {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    System.out.println("Enter a Sudoku puzzle:");
    int[][] grid = new int[9][9];
    for (int i = 0; i < 9; i++)
      for (int j = 0; j < 9; j++)
        grid[i][j] = input.nextInt();

    return grid;
  }

  /** Obtain a list of free cells from the puzzle */
  public static int[][] getFreeCellList(int[][] grid) {
    // Determine the number of free cells
    int numberOfFreeCells = 0;
    for (int i = 0; i < 9; i++)
      for (int j = 0; j < 9; j++)
        if (grid[i][j] == 0)
          numberOfFreeCells++;

    // Store free cell positions into freeCellList
    int[][] freeCellList = new int[numberOfFreeCells][2];
    int count = 0;
    for (int i = 0; i < 9; i++)
      for (int j = 0; j < 9; j++)
        if (grid[i][j] == 0) {
          freeCellList[count][0] = i;
          freeCellList[count++][1] = j;
        }

    return freeCellList;
  }

  /** Print the values in the grid */
  public static void printGrid(int[][] grid) {
```

```
    for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++)
        System.out.print(grid[i][j] + " ");
      System.out.println();
    }
  }

  /** Search for a solution */
  public static boolean search(int[][] grid) {
    int[][] freeCellList = getFreeCellList(grid); // Free cells
    if (freeCellList.length == 0)
      return true; // "No free cells");

    int k = 0; // Start from the first free cell
    while (true) {
      int i = freeCellList[k][0];
      int j = freeCellList[k][1];
      if (grid[i][j] == 0)
        grid[i][j] = 1; // Fill the free cell with number 1

      if (isValid(i, j, grid)) {
        if (k + 1 == freeCellList.length) { // No more free cells
          return true; // A solution is found
        }
        else { // Move to the next free cell
          k++;
        }
      }
      else if (grid[i][j] < 9) {
        // Fill the free cell with the next possible value
        grid[i][j] = grid[i][j] + 1;
      }
      else { // free cell grid[i][j] is 9, backtrack
        while (grid[i][j] == 9) {
          if (k == 0) {
            return false; // No possible value
          }
          grid[i][j] = 0; // Reset to free cell
          k--; // Backtrack to the preceding free cell
          i = freeCellList[k][0];
          j = freeCellList[k][1];
        }

        // Fill the free cell with the next possible value,
        // search continues from this free cell at k
        grid[i][j] = grid[i][j] + 1;
      }
    }
  }

  /** Check whether grid[i][j] is valid in the grid */
  public static boolean isValid(int i, int j, int[][] grid) {
    // Check whether grid[i][j] is valid at the i's row
    for (int column = 0; column < 9; column++)
```

```
        if (column != j && grid[i][column] == grid[i][j])
          return false;

    // Check whether grid[i][j] is valid at the j's column
    for (int row = 0; row < 9; row++)
      if (row != i && grid[row][j] == grid[i][j])
        return false;

    // Check whether grid[i][j] is valid in the 3 by 3 box
    for (int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++)
      for (int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++)
        if (row != i && col != j && grid[row][col] == grid[i][j])
          return false;

    return true; // The current value at grid[i][j] is valid
  }

  /** Check whether the fixed cells are valid in the grid */
  public static boolean isValid(int[][] grid) {
    for (int i = 0; i < 9; i++)
      for (int j = 0; j < 9; j++)
        if (grid[i][j] < 0 || grid[i][j] > 9 ||
           (grid[i][j] != 0 && !isValid(i, j, grid)))
          return false;

    return true; // The fixed cells are valid
  }
}
```

***<Output>***

```
Enter a puzzle:
0 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0

The solution is found:
9 6 3 1 7 4 2 5 8
1 7 8 3 2 5 6 4 9
2 5 4 6 8 9 7 3 1
8 2 1 4 3 7 5 9 6
4 9 6 8 5 2 3 1 7
7 3 5 9 6 1 8 2 4
5 8 9 7 1 3 4 6 2
3 1 7 2 4 6 9 8 5
6 4 2 5 9 8 1 7 3
```

***<End Output>***

The program invokes the readAPuzzle() method (line 6) to read a Sudoku puzzle and return a two-dimensional array representing a Sudoku grid. There are three possible outputs from the program:

- The input is invalid (line 9)
- A solution is found (line 11).
- No solution is found (line 15).

***<Side Remark: getFreeCellList method>***

The getFreeCellList(int[][] grid) method returns a two-dimensional array storing the free cell positions. freeCellList[i][j] indicates a free cell at row index i and column index j. The method first counts the number of free cells (lines 35-39), then creates the array for storing free cell positions (lines 42-49).

***<Side Remark: search method>***

The search(int[][] grid) method starts search from the first free cell with k = 0 (line 69), where k is the index of the current free cell being considered in the free cell list, as shown in Figure 5.

The value in a free cell starts with 1 (line 74). If the value is valid, the next cell is considered (line 81). If the value is not valid, its next possible value is considered (line 86). If the value is already 9, the search is backtracked (lines 89–97). All the backtracked cells become free again and their values are reset to 0 (line 93). If the search backtracks to the free cell list at position k and the current free cell value is not 9, increase the value by 1 (line 101) and the search continues.

The search method returns true when the search advances but no more free cells are left (line 78). A solution is found.

The search returns false when the search is backtracked to the first cell (line 91) and all possible values are exhausted for the cell. No solution can be found.

***<Side Remark: isValid method>***

The isValid(i, j, grid) method checks whether the current value at grid[i][j] is valid. It checks whether grid[i][j] appears more than once at row i (lines 109-111), at column j (lines 114–116), and in the 3×3 box (lines 119–122).

How do you locate all the cells in the same box? For any grid[i][j], the starting cell of the 3×3 box that contains it is grid[(i / 3) * 3][(j / 3) * 3], as illustrated in Figure 6.

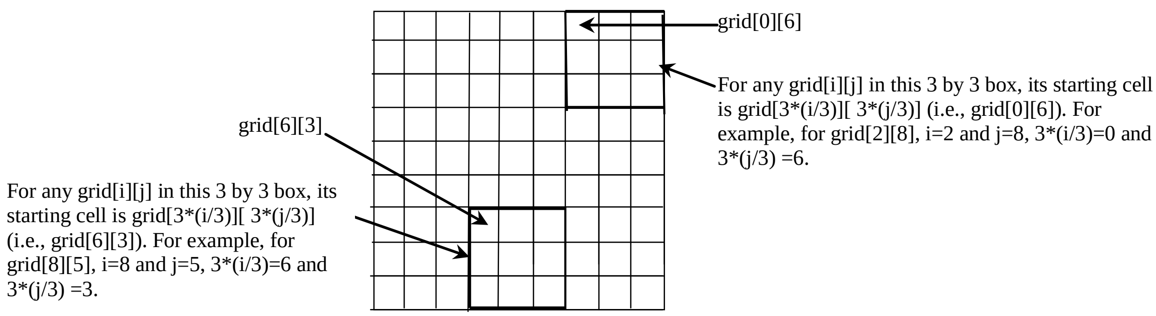



**Figure 6**

*The location of the first cell in a 3×3 box determines the locations of other cells in the box.*

With this observation, you can easily identify all the cells in the box. Suppose grid[r][c] is the starting cell of a 3×3 box, the cells in the box can be traversed in a nested loop as follows:

```
// Get all cells in a 3 by 3 box starting at grid[r][c]
for (int row = r; row < r + 3; row++)
  for (int col = c; col < c + 3; col++)
    // grid[row][col] is in the box
```

***<Side Remark: find one solution>***

Note that there may be multiple solutions for an input. But the program will find one such solution. You may modify the program to find all solutions in Programming Exercise 7.27.

***<Side Remark: input file>***

It is cumbersome to enter 81 numbers from the console. When you test the program, you may store the input in a file, say sudoku.txt, and run the program using the following command:

```
java Sudoku < sudoku.txt
```

This program is not user friendly. You can improve it by providing a GUI interface. See Exercise 18.25.