COM6516: Final assignment (60%)

Adam Funk

18 December 2017

1 Task

Develop a program with a GUI and a text-file reader that solves sudoku puzzles, with the characteristics described below.

The due date is 15 January 2018 at 17:00 GMT (UK time, as indicated on MOLE).

1.1 Specification

- The GUI should include the following:
  - buttons for loading a text file, starting the solver, interrupting it, clearing all the cells, and quitting the program;
  - a grid of 81 cells arranged in a square, with visible boundaries between the cells;
  - a "status console" showing status output from the program.

  The user can edit cells with the mouse and keyboard when the solver is not running, but the cells should be read-only (to the user) while it is running. Cells that are solved (reduced to one possible value) or impossible (because of an inconsistency in the puzzle input) must be colour-coded yellow or red, respectively.

  Figure 1 shows a satisfactory example, but your program does not need to look exactly like this.

- The status console should be visibly separate from the puzzle grid (a different background colour is sufficient) and should have a fixed number of lines. When a new message is added to a full status console, the oldest message should "roll" off the top. A message should appear when any of the following happens:
  - the solving procedure starts;
  - the procedure gets stuck (it detects that it cannot eliminate any more possibilities with the information available);
  - the procedure succeeds (solves the puzzle);

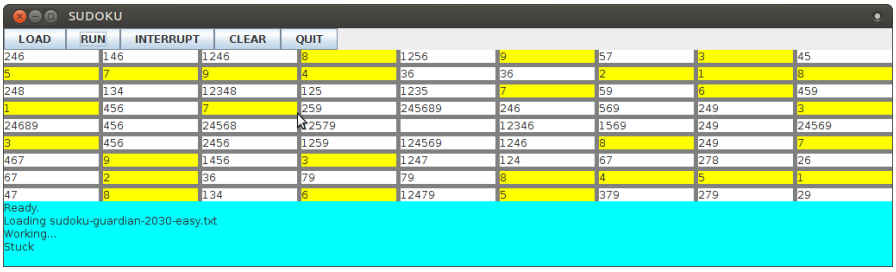

Figure 1: Example of a "stuck" program

  - the procedure detects an inconsistency or impossibility in the puzzle;
  - the program is loading a file;
  - the user interrupts the solving procedure.

- The solving procedure should try to update cell displays whenever it can and should have occasional very short pauses (e.g., using `Thread.sleep`) so the user can see it progress instead of just instantly getting the solution). Updating cell displays will need to use the `invokeLater` technique in the Week 10 lecture.

  Once the solving procedure starts, each cell should have the following characteristics.

  - It is not user-editable.
  - It is blank if the program has not eliminated any of the 9 possibilities yet.
  - It contains one digit on a yellow background if the cell has been solved.
  - It contains 2 to 8 digits on a white background if the cell has not been solved but some values have been eliminated.
  - It has a red background if an inconsistency has been detected, such as the following:
    * it has only one possible value and another cell in its group (row, column, or block) also has been reduced to the same value;
    * it has no possible values because all 9 digits have been used up in its group.

- The solving procedure should be able to determine when it has run through all the possible avenues for solving the puzzle without making any progress; in this case it should stop looping and report to the status console that it is stuck.

- The listener for the *solve* or *run* button will need to start the solving procedure in a separate thread to avoid locking the GUI, and it will need to check repeatedly whether the *interrupt* button has been pressed. (See the Week 10 lecture.)

- Your program only needs to handle the most common 9×9 standard sudoku format with the digits 1–9.
- You can use AWT instead of Swing if you wish.

1.2 File input

The file-loading code in your program should accept a plain-text file such as the following.

```
___8_9_3_
579____1_
_____7_6_
1_7_____3
_________
3_____8_7
_9_3_____
_2____451
_8_6_5___
```

- Loading a file should overwrite all cells in the puzzle.
- The program can ignore extra lines if the file contains more than 9.
- The program can ignore extra characters if a line contains more than 9.
- Any character other than a decimal digit can be treated as a blank cell.
- If any line contains fewer than 9 characters, or the file contains fewer than 9 lines, your program must report an error to the status console and continue running.
- If any `java.io.*` classes or their methods throw an exception, your program must report an error to the status console and continue running.

1.3 Hints and suggestions

The following link contains useful information about methods for solving sudoku puzzles. You can use any of them, but in my experience item 3 (*pencilling in*) works well with a computer model of the puzzle. Your program will, however, have to use some other techniques (probably *pairs* and *triples* using this page's terminology) to solve more difficult puzzles (for full marks).

`http://www.paulspages.co.uk/sudoku/howtosolve/`

You can represent the puzzle data as 27 cell groups, each containing 9 cells (9 groups for the rows, 9 for the columns, and 9 for the square blocks; each cell will be a member of one of each type of group). Each cell can be displayed on screen with a `JTextArea`.

The cell will need several fields for such things as the possible values it can have, the text to display, whether it has been solved yet, and whether it has been marked as impossible to solve.

1.4 Loading the file

- Example of using a `JFileChooser`:

  `https://www.mkyong.com/swing/java-swing-jfilechooser-example/`

- Examples of reading a file line-by-line:

```
BufferedReader br = new BufferedReader(new FileReader(file));
String line;
while ((line = br.readLine()) != null) {
  // process the line.
}
```

  or

```
BufferedReader br = new BufferedReader(new FileReader(file));
for (String line; (line = br.readLine()) != null; ) {
  // process the line.
}
```

  You need to modify either of these examples to use `try` and a `finally` block around `br.close()`.

- The easiest way to see if a one-character `String` represents an integer and, if so, to convert it, is to catch the `NumberFormatException` that `Integer.parseInt` can throw:

```
try {
  int i = Integer.parseInt(line.substring(c, c+1));
  // store the value of i in the cell and mark it solved
}
catch (NumberFormatException e) {
  // mark the cell as having no specified value yet
}
```

2 Rules

2.1 Unfair means

This is an individual assignment so you are expected to work on your own. You may discuss the general principles with other students but you must not work together to dev elop your solution. You must write your own code. All code submitted may be examined using specialized software to identify evidence of code written by another student or downloaded from online resources.

2.2 Submission

Upload your `*.java` files to MOLE using the assignments button on the menu bar. Do not submit any other files, such as `*.class` or `*.jar` files. (Ignore the week 1 handout about using ZipCentral.) This assignment does not require the `sheffield` package or any libraries other than the Java standard library, and no other libraries will be used when your code is compiled and executed for marking.

Your code should compile and run under Java 1.8 on the command line:

```
javac *.java
java NameOfMainClass
```

You can submit your work more than once; the last version submitted will be marked, so check it carefully. You will not be able to submit any code after the final deadline.

If you are unable to upload your code to MOLE for technical reasons, e-mail `a.funk@sheffield.ac.uk` **in advance** to make arrangements for submitting it by e-mail.

3 Marking

Marks and feedback will be returned through MOLE within 3 weeks.