Introduction to the Assignment

In this lab, you will create a program to allow a user to solve Sudoku puzzles. In writing this program you will gain experience reading from files and working with exception handling.

Sudoku is a popular puzzle in which the player is given a 9 x 9 grid, which can also be viewed as a 3 x 3 grid, where each entry in the grid is itself a 3 x 3 grid. Each entry can hold a number from 1 to 9. Initially, a grid is shown with some numbers filled in. The player’s goal is to fill in the empty squares such that each row contains the numbers 1 through 9, each column contains the numbers 1 through 9 and each 3 x 3 square contains the numbers 1 through 9. If you are unfamiliar with this puzzle, I recommend that you try Websudoku (http://www.websudoku.com/) to get a better understanding.

Here are some snapshots from my implementation of Sudoku:

This is the starting screen. The grid is drawn and the partial solution is displayed in blue. The user must fill in the empty squares. The user cannot change the values in the squares provided by the program. Notice that the “Check” and “Reveal solution” buttons are enabled while “Next game” is disabled.
Here the user has entered some values in squares. Notice that the user’s entries are in black.

If the user clicks the Check button and the puzzle is incomplete or not solved correctly, this message pops up.

If the user clicks the Check button and the puzzle is solved correctly, this message pops up. After hitting OK, the “Check” and “Reveal solution” buttons should be disabled and “Next game” should be enabled.
If the user clicks the “Reveal solution” button all blank and incorrect squares are filled in, using a blue font. “Next game” is enabled and the other buttons are disabled.

When the user clicks “Next game” the starting grid for the next puzzle is displayed.

When the user clicks “Next game” if all known puzzles have been solved, this message pops up. After hitting ok, all the buttons are disabled.
The Program Design

This project will have 4 classes. These classes are:

- **Sudoku** - this is the class that sets up the user interface and contains the main method.
- **Sudoku3By3** - this is one of the 3 x 3 squares in the grid. It manages the user interface of a small square of grids. The Sudoku class will have 9 of these to make the entire grid.
- **SudokuSolution** - this class holds the correct solution. It will be responsible for reading the solution from the file and helping to determine if a user’s answer is correct.
- **SolutionBadException** - this is an exception class that you will create to help deal with problems that arise if the solution file has the wrong format.

We have not talked about packages in class. I encourage you to read Section 1.7 of the textbook (or Section 3.6 if you have the old version of the text) for a description of packages. In this and all future labs, we will place classes within packages. For this lab, you will have 1 package named “sudoku” (by convention, package names begin with small letters) and all four of these classes will be within that package.

You will also be using some classes and interfaces that Java provides that you are not yet familiar with. These are described briefly below.

**JOptionPane** - This is the class that you use to create pop-up messages. In particular, you will call the method:

```
JOptionPane.showMessageDialog(gamePanel, "Congratulations!");
```

to create a pop-up dialog with the message “Congratulations!”.

**BorderFactory** - This allows you to draw the red border around the 3 x 3 squares. To do this, you will call:

```
squarePanel.setBorder(BorderFactory.createLineBorder(Color.RED, 3));
```

This would draw a red border around square Panel with a thickness of 3 pixels.

Writing the Program

Create a new Eclipse project called Sudoku. Create a package named "sudoku". You can do this by clicking on the icon in the toolbar at the top of the Eclipse window. When you construct your classes, be sure that the package is listed as “sudoku” on the window that comes up, as shown at the right. This will insert the line:

```
package sudoku;
```
at the top of the file, before the import statements.

You should download the solution files from the website and import them into your Eclipse project. These should not go inside the src folder. These are the files named
solution1.txt, solution2.txt, etc.

**Step 1: Construct the user interface**

Laying out the buttons should be very similar to what you did for the ImageManipulator lab, so we will focus on what you need to do to draw the Sudoku grid.

Java provides a GridLayout class that allows you to create displays where the display is broken into equally-sized areas using a specific number of rows and columns. Looking at the demo or the screen shots from earlier, you should see that the screen consists of 3 rows and 3 columns, each identified by a red border. Thus, at the top level, we have a 3 x 3 grid. Within each of the red squares is another, smaller 3 x 3 grid, which is where the numbers actually appear. Each entry of the inner grid contains a JTextField where the user can enter their answers or the game can display its hints.

By default, JPanels use FlowLayout. You can tell a JPanel to use a GridLayout instead, like this:

```java
myPanel.setLayout (new GridLayout (numRows, numCols));
```

So, first have your main Sudoku panel use a 3 x 3 grid. Then, create another class, Sudoku3x3, that also contains a 3 x 3 grid. In each of these grids, add a JTextField. Your Sudoku class should construct 9 instances of Sudoku3x3 and place each of them in the main grid.

When you add a component to a grid, you do not need to indicate where in the grid it will go, Java will just fill up the grid from left to right and top to bottom, like words on a page. In this way, it is a lot like FlowLayout, except that each entry has the same size square and Java draws the grid.

To get the red border, you need to call the setBorder method on the component (in this case a panel) that you want to put the border around.

```java
public void setBorder(Border border)
```

To get a Border object, you call one of the methods in the BorderFactory class. For a border as in the demo, you would say:

```java
Border myBorder = BorderFactory.createLineBorder (Color.RED, 3);
```

This creates a border that is drawn as a simple red line that is 3 pixels wide. You would then pass this border as the parameter in your setBorder call.

You can produce many other types of borders with BorderFactory. Look at the online Java API and feel free to experiment if you like.

To center the text that appears in the JTextFields, use

```java
public void setHorizontalAlignment (int alignment)
```

passing JTextField.CENTER as the alignment parameter.

To change the font used in a JTextField, call:

```java
public void setFont (font f)
```

creating the fonts the same way as you did for the IDCard lab. To change the color of the font, call:

```java
public void setForeground (Color foreground Color)
```

Note: I have sometimes seen a stack trace display when I start my program in which none of my methods appear. This seems to be related to changing the font. It does not always
occur and it does not seem to affect the game at all. So, if you get a stack trace and none of the methods on the trace are ones you wrote, it is probably safe to ignore.

**Step 2: Putting numbers on the grid**

The user may think of the Sudoku grid as 3 x 3 blocks, with each block containing a grid of 3 x 3 numbers. For much of your program, though, it will be easier to treat it as a single 9 x 9 grid. However, we have just created a representation that mirrors the user’s view. We will therefore need a way to figure out which JTextField we are referring to when we want to know what value is in row 2, column 5 for example. Here, we see a 1 in that square. So, where is that? First, we need to determine which cell of the outer grid it is in. That is row 0, column 1. Then we need to determine which row and column it is in in the inner grid, considering just that 3 x 3 grid. There it is row 2, column 2.

Here, we see a 1 in that square. So, where is that? First, we need to determine which cell of the outer grid it is in. That is row 0, column 1. Then we need to determine which row and column it is in in the inner grid, considering just that 3 x 3 grid. There it is row 2, column 2.

How do we do this mapping? Let’s first focus on the rows. Row 0 of the outer grid covers rows 0-2 of the overall grid. Row 1 covers rows 3-5 and Row 2 covers rows 6-8. So, to determine which row of the outer grid we need, we can take the row number and divide by 3, using integer arithmetic, so we drop any fractional part. In this case 2 / 3 = 0. So, we know that the row that is row 2 overall is in row 0 of the outer grid.

We could do the same thing for the columns and determine that we are interested in column 1 of the outer grid (5 / 3 = 1). In this way, we could identify a single red square. Now, how can we figure out which JTextField is involved? Here we can use the modulus operator (often just called mod, and written using % in Java). mod does the integer division, but this time it evaluates to the remainder and throws away the quotient. This will identify the position of the JTextField we want within the red square.

So, 2 % 3 = 2. (The remainder after integer division is 2.) And 5 % 3 = 2 (again because the remainder after integer division is 2). So, the JTextField is in row 2, column 2 of the red square located at row 0, column 1.

You should write a setValue method in Sudoku that takes a row number and column number for the overall grid (like row 2, column 5) as well as a value. It should determine which Sudoku3x3 object is in that position and call a setValue method that you define in the Sudoku3x3 class. This should also take a row number, column number and value, but this time the row number and column number must range between 0 and 2. The Sudoku3x3 method should find the JTextField in that part of the grid and call setText to display the value.

Modify your main method to test this by adding some calls to your Sudoku setValue method and see if the correct cell in the grid gets set. It would be a good idea to try this on the four corners as well as some cell not at an edge.

The values that get set with this method are the hints that your program will provide to the user. You should therefore pick a font that makes them look different than the ones the user enters. You will also want to prevent the user from changing those values. To do that, call:

```java
public void setEditable (boolean b)
```

passing false as the parameter. This will prevent the user from modifying those fields.
Step 3: Reading a solution from a file

To determine if the user has solved the puzzle, you will be reading the solution from a file along with the location of where to display numbers when the game starts. The contents of the file is formatted as follows:

- Each row of the solution is displayed on a single line, with each number separated by a single space.
- After all 9 rows are displayed, there is a line containing a single '@'.
- Following that are a number of rows containing a row number and column number identifying which values should be displayed to the user when the game starts.

For example, on the left is the contents of a file and on the right is a picture of the starting screen.

```
1 8 6 5 2 3 9 4 7
2 9 7 4 1 8 3 6 5
4 5 3 6 7 9 8 2 1
9 7 1 8 6 5 4 3 2
8 4 5 3 9 2 1 7 6
3 6 2 7 4 1 5 8 9
5 2 4 1 3 6 7 9 8
7 1 9 2 8 4 6 5 3
6 3 8 9 5 7 2 1 4
@
0 1
0 2
0 5
1 2
1 3
1 4
1 6
1 7
1 8
2 3
2 5
2 6
2 7
2 8
3 1
3 2
3 3
3 4
3 5
3 7
4 1
4 7
5 0
5 3
5 4
```
Solutions are stored in files named “solution1.txt”, “solution2.txt”, etc. You can look at these by double-clicking on them in Eclipse.

You should create a class called SudokuSolution. The constructor for this class should be passed the name of a file containing a solution. It should read the first 9 rows from the file and save it in a 2-dimensional 9 x 9 array.

In writing the code to read the solution from the file, you will see a compiler error message like this: “Unhanded exception type FileNotFoundException.” This is a warning to you that you need to figure out what to do if the file you try to open does not actually exist. You will need to put the call to the SudokuSolution constructor into a try block. In the corresponding catch block, you should handle the FileNotFoundException by displaying an error message to the user in a pop-up dialog box. To display a message in a pop-up window, say:

```
JOptionPane.showMessageDialog (this, “Your message goes here.”);
```

You should also disable all 3 buttons so that the user can’t try to play the game. To disable a button, call this method on each button:

```
public void setEnabled (boolean b)
```

Passing false in will cause the button to be dimmed and no longer react to the user clicking on it, until you call setEnabled again with true as the parameter.

SudokuSolution should also define a method with this signature:

```
public int getCorrectValue (int row, int col)
```

This method should be passed a value between 0 and 8 for each of the row and column and return the value in the solution at that location.

Now add an action listener to the “Reveal solution” button. This action listener should walk the entire 9 x 9 grid, get the correct value from the solution and use the setValue methods from step 2 to display the solution on the grid. Run your program and make sure that the solution displayed to the user is the same as what the file contains.

**Step 4: Displaying hints to the user**

Next, you should read the pairs of values that correspond to the hints that should be displayed. Modify the SudokuSolution constructor to also read these values from the file. Note that in this case, you do not know how many lines to read from the file. You need to just read until there are no more ints left in the file.
When you read in a row, column pair, you should call the `setValue` method defined in Sudoku so that the hint is displayed to the user. To do this, you will need to pass the Sudoku object to the `SudokuSolution` constructor so that it can make that call.

**Step 5: Making the file reading robust**

One problem with reading files is that various errors might exist in the file. In step 3, you dealt with the problem of the file not existing. Now, consider various ways in which the contents of the file might be different than what you expect. For example,

- The file might have fewer than 9 rows defining the solution
- The file might have more than 9 rows
- A row might have more or fewer than 9 columns
- A number in the solution might be something other than 0 through 8. It might even be a letter.
- Similarly, the hints might not be what you expect.

Think about all the different ways that the file might be formatted incorrectly and modify the code that reads from the file to identify these errors. Create a new class `SolutionBadException` as a subclass of `Exception`. If you find any errors in the file contents, construct an instance of this new exception class and throw it. The `Sudoku` class should handle this exception by again displaying an error message that is suitable for the user and disabling the buttons.

Test this by creating more solution files but introducing errors into them.

**Extra Credit Step 6 (Moderate): Checking the user’s answers**

At this point, you do not really have enough to let the user play the game, but you have done the work of reading from files and working with exception handling, which are the main goals of this lab. The remaining steps allow you to complete the game and are worth extra credit.

To check the user’s answer, you need a pair of methods called `getValue` in your `Sudoku` and `Sudoku3x3` classes to find out what value a user types in a `JTextField`. These do the same types of arithmetic as the `setValue` methods to determine which `JTextField` to read a value from. To get the value from a `JTextField`, call:

```java
public String getText()
```

If the `JTextField` is empty, this will return the empty string: `""`. You will need to check for this special case and return a special value, like 0 in that case. If it is not empty, it will return the number as a String `"1"`, not the int 1. To convert the String to an int, say:

```java
return Integer.parseInt(myString);
```

Add an action listener to the Check button that walks through the grid checks the user’s answer. If every cell is correct, display a “You won” pop-up message. If there are any mistakes, display some other message.

**Extra Credit Step 7 (Simple): Next game**

I am providing multiple solution files, all named in sequence “solution1.txt”, “solution2.txt”, etc. When the user clicks the next game button, you should clear the grid and start over with the next solution file.

You should also update your code to enable and disable buttons as follows:

- At the start and when a new game is loaded, the “next game” button should be disabled.
- If the user solves a puzzle or reveals the solution, “next game” should be enabled and the other 2 disabled.
• When the file is not found, all the buttons should be disabled and you should display a message in a pop-up window telling the user that the user has solved all of the puzzles.

**Grading**

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Step 1: User interface</td>
</tr>
<tr>
<td>15</td>
<td>Step 2: Putting numbers on grid</td>
</tr>
<tr>
<td>15</td>
<td>Step 3: Revealing solution</td>
</tr>
<tr>
<td>10</td>
<td>Step 4: Displaying hints</td>
</tr>
<tr>
<td>10</td>
<td>Step 5: Robust file reading</td>
</tr>
<tr>
<td>15</td>
<td>Comments (including javadoc comments) and assert statements</td>
</tr>
<tr>
<td>20</td>
<td>Style</td>
</tr>
<tr>
<td>5, 10</td>
<td>Extra credit (simple, moderate respectively)</td>
</tr>
</tbody>
</table>

**Turning in Your Work**

Create a jar file, being sure to include your source code. Then, go to ella. Click on COMSC 102 in the toolbar menu across the top. Then click on Assignments in the left column. Click the submit link for this assignment. Click the Add Attachment button. Use the Browse... button to upload a local file. You should submit your jar file. Click Continue. Then click the Submit button.