Advanced Programming Coursework

Student ID - 000540476

3/20/2010

Toby Gates

Table of Contents

1 – Statement of Progress for Part One

For Part 1 of the Coursework, I have completed Sections (a), (b), (c) and (d). However, I did make an attempt at solving Section (e) after setting the amount of Passengers so they weren't a multiple of the Helicopter Capacity. Eventually, I did not manage to fully solve (e) as I did for the other tasks; the following section shows screenshots and contains descriptions on which sections of Part 1 were completed for the program.

2 - Part One

Section A

Output produced by the Program for Section A

```
compile-single:
run-single:
Passenger 1 boards helicopter
Passenger 2 boards helicopter
Passenger 3 boards helicopter
Passenger 4 boards helicopter
Passenger 5 boards helicopter
Flight takes off
Flight lands
All Passengers get out of helicopter
BUILD SUCCESSFUL (total time: 6 seconds)
```

What basically take place inside the program is the boarding of Passengers onto a single Helicopter which then takes off and flies to a destination.

After a delay that occurs after the takeoff (used to indicate the flight time of the helicopter) the Helicopter then lands at the destination and allows the passengers to get off of the helicopter.

Since there is only a single thread of control, this means only the first five passengers can get on the Helicopter due to the helicopter having a maximum capacity of 5.

Section B

Changes made to the Code for Section B

```
public class Passenger {  public class Passenger extends Thread {
```

BEFORE

The screenshot above shows how the class Passenger was before the change made to the code, it never used to extend or implement anything else.

AFTER

The screenshot above shows the class Passenger after this section of the code was changed. As you can see the class Passenger now extends the class Thread.

```
        // Run the Passengers (they will try to board the helicopter)
        for (Passenger p : passengerQueue) {
            p.run();
        }
    }
}
```

BEFORE

The screenshot shows the calling of the run() method for the passengers before any changes were made to the code, the run() method is located in the Passenger class.

```
        // Run the Passengers (they will try to board the helicopter)
        for (Passenger p : passengerQueue) {
            //p.run();
            p.start();
        }
    }
}
```

AFTER

The screenshot above shows the changes made to the code. Here I have commented out the call to the run() method in the passenger class and replaced it with the start() method (since the code was changed to allow the class Passenger to extend the class Thread) which when called will split the single thread of control into multiple threads for each passenger.

Output produced by the Program for Section B

When the program is run; the start() method is called for the passengers in the "RunHelicopterFlights" class.

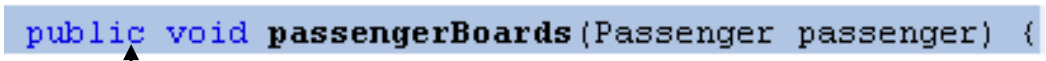
This causes all 15 passengers attempt to board the helicopter simultaneously due to the fact that the start() method caused the single thread of control to split into multiple threads for each passenger.

The problem the output demonstrates is the fact that all 15 passengers can board the Helicopter despite the Helicopter having a constant maximum capacity of 5.

```
compile-single:
run-single:
Passenger 2 boards helicopter
Passenger 4 boards helicopter
Passenger 6 boards helicopter
Passenger 8 boards helicopter
Passenger 10 boards helicopter
Passenger 12 boards helicopter
Passenger 1 boards helicopter
Passenger 14 boards helicopter
Passenger 3 boards helicopter
Passenger 5 boards helicopter
Passenger 7 boards helicopter
Passenger 9 boards helicopter
Passenger 11 boards helicopter
Passenger 13 boards helicopter
Passenger 15 boards helicopter
Flight takes off
Flight lands
All Passengers get out of helicopter
BUILD SUCCESSFUL (total time: 5 seconds)
```

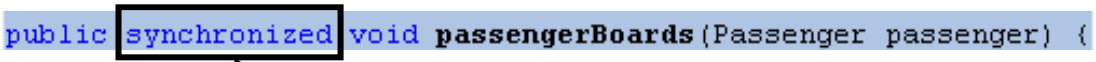
Section C

Changes made to the Code for Section C



BEFORE

The screenshot above shows the method declaration for the passengerBoards() method in the Helicopter class before any edits were made to it for Section C of Part One.



AFTER

The screenshot above shows the method declaration for the passengerBoards() method after the change was made.

The keyword synchronized was added to the method declaration which provides a lock on the method when a thread accesses it, no other threads can access it when one thread is accessing it until the lock is released.

How the change in the Output was caused for Section C

The output was changed due to the fact that the first 5 passenger threads that managed to access the method when there was no lock active are displayed as an output. The lock provided by the keyword synchronized prevents all 15 passenger threads from accessing the passengerBoards() method at the same time in one big rush.

Only 5 passengers are shown in the output again, this is due to the IF statements in the passengerBoards() method making use of the HELICOPTER_CAPACITY constant to call the takeOff() method if the amount of passengers is equal to the HELICOPTER_CAPACITY (which is 5).

Section D

Code for both of the Methods after the changes were made

```
public synchronized void passengerBoards(Passenger passenger) {

    //start of while loop used to get passengers boarding to wait.
    while (getNumberOfPassengers() == HELICOPTER_CAPACITY) {
        try {
            wait();
        } catch (InterruptedException ex) {
            System.out.println(ex);
        }

    }

        //start of if loop allowing the passengers to get on the plane.
        if (getNumberOfPassengers() < HELICOPTER_CAPACITY) {
           System.out.println("Passenger " + passenger.getNumber() +
                   " boards helicopter");
           delayWhilePassengerSitsDown();
           passengers.add(passenger);
        }

        //start of another if loop, which is seperate from the while loop,
        //which allows the Helicopter to take off when it is full.
        if (getNumberOfPassengers() == HELICOPTER_CAPACITY) {
            takeOff();
        }
}
```

The screenshot to the left displays the code for the synchronized method called "passengerBoards" in the Helicopter class.

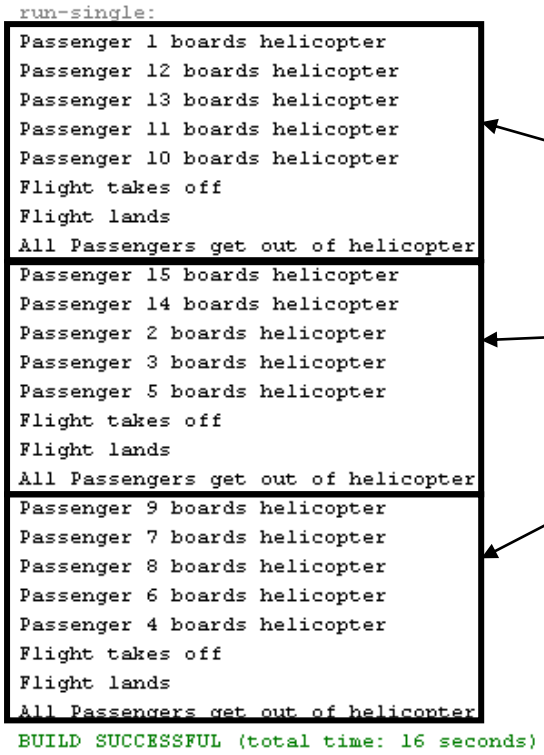
The changes shown are the ones applied in order to achieve Section D for Part 1.

```
private synchronized void allPassengersGetOut() {

    passengers.clear();
    System.out.println("All Passengers get out of helicopter");

    //notifies the waiting threads that the helicopter is free.
    notifyAll();
}
```

The screenshot above shows the synchorized method of allPassengersGetOut() which causes the passengers to get out of the plane once it has landed.

After an indication has been display showing that All of the Passengers have gotten out of the Helicopter; the notifyAll() is called in order to allow the other passengers (threads) to access the passengerBoards() method to allow them to get on the plane.

Output produced by the Program for Section D



The image to the left shows the Output of the program for Section D.

The output shows all 15 passengers boarding 3 separate flights due to the Helicopter having a capacity of 5 passengers.

Section E

Problem Observed when Number of Passengers is not a multiple of HELICOPTER_CAPACITY

After I had changed the Number of Passengers so they are not a multiple of the constant HELICOPTER_CAPACITY (which is 5) for Section E, the problem I observed was that the plane doesn't take off and fly anywhere due to the fact that it doesn't reach the helicopter's maximum capacity (5 passengers).

So the last two passengers get on board the flight (due to changing the number of passengers from 15 to 17) and don't get anywhere.

```
run-single:
Passenger 1 boards helicopter
Passenger 17 boards helicopter
Passenger 16 boards helicopter
Passenger 15 boards helicopter
Passenger 14 boards helicopter
Flight takes off
Flight lands
All Passengers get out of helicopter
Passenger 3 boards helicopter
Passenger 5 boards helicopter
Passenger 2 boards helicopter
Passenger 4 boards helicopter
Passenger 6 boards helicopter
Flight takes off
Flight lands
All Passengers get out of helicopter
Passenger 12 boards helicopter
Passenger 13 boards helicopter
Passenger 10 boards helicopter
Passenger 11 boards helicopter
Passenger 9 boards helicopter
Flight takes off
Flight lands
All Passengers get out of helicopter
Passenger 7 boards helicopter
Passenger 8 boards helicopter
BUILD SUCCESSFUL (total time: 16 seconds)
```

After making a couple of attempts at Section E, I decided to stop working on it as I really needed to push on with working on the JavaBeans sections of Part 2 of the coursework. So eventually, I didn't find a solution to it with the time I had available to spend working on it.

3 – Statement of Completed Stages for Part Two

For Part 2 of the Coursework I have completed Stage 1. The following enhancements I made to the program for Stage 1 are as follows.

Improvements to the Look and Behaviour

The improvements to the look and behaviour I had implemented onto the program were;

- The Validation of the Input for the Number of Blanks Entered.
- Improvements to the Overall Look and Feel of the Program (this includes the use of colour, creation of design and the use of tooltips on certain objects).
- A message to indicate to the user when the puzzle has been successfully completed, which is displayed as a message on screen.

Additional Features

The Additional Features I had implemented or attempted to implement were;

Completed

- Allowing a partially completed puzzle to be saved for later.

Attempted

- Allow puzzles to be loaded from a file.
- Allowing a puzzle from the program to be printed off.

Additional Feature of your own that is not in the Specification

- Allowing the user to display the rules and instructions on how to play Sudoku in a text file.

Design Pattern

The Design Pattern I had implemented into the program was MVC. Whereas the Puzzle and Puzzle Generator classes were the Model classes of MVC for the program, the Sudoku View and Cell Note View were the View Classes of MVC for the program with a mock up of a Controller class called Sudoku Controller just to demonstrate that I know how MVC works.

I would've attempted to create a full Controller class, but I didn't want to mess up the functionality of the program completely and I didn't have enough time to do actually do it with other coursework and presentations taking priority.

I have also completed Stage 2 and Stage 3 of the coursework, which are the creation of a JavaBeans version of the Sudoku program and creation of another application to demonstrate the JavaBean working.

For Stage 2, I managed to implement my JavaBean onto the Sudoku program instead of using the separate components that make up the JavaBean (see screenshot in the last section of this report).

For Stage 3, I just made a blank basic program with a JButton and a JTextArea just to show that my JavaBean can be loaded with NetBeans and its functionality can be used in another application if necessary (see screenshot in the last section of this report).

4 – List of JavaBeans implemented in Stage 2 for Part Two

The information of my JavaBean I have implemented is shown below;

Class Name

SudokuSelectBean.java

Description

This Bean allows the user to increment and decrement the integer value displayed in the JTextField (you can also type an Integer value). It also allows the flexibility of not including validation, in order for people to set up their own according to any different programs they decide to use it in.

Features of JavaBeans implemented

Made use of a BeanInfo file called "SudokuSelectionBeanBeanInfo.java" to give the Bean an Icon of an orange arrow.

Made use of a BeanInfo file called "SudokuSelectionBeanBeanInfo.java" to restrict some of the properties using a PropertyDescriptor.

5 – List of Bugs or Weaknesses in my Work

1. The main weakness of my work is the fact that the JavaBean I created only seems to get the default value it is set to when it is called from an application or class even when the JavaBean is not set to the default value. The getValue() method in the bean itself however, works fine as I have tested it and it works.

2. Some other weaknesses in my works are the fact that I started the Print Puzzle and Load Puzzle features for the original Sudoku program, but I ran out time when it came to getting them finished. So their buttons are available there with the commented out code I was using to develop these features.

3. A bug in the program is the fact that a puzzle can still be generated if an incorrect value is entered and the Get Puzzle button is clicked on, the validation messages will display however the puzzle will still show up after they are closed.

6 – Description of any special strengths of my JavaBean(s)

1. One strength is allows the user to be able to set up an incrementing and decrementing number field JavaBean for various flexible uses, such as selecting and loading the page of a book or a file set to a corresponding number using a case statement. It can also be used for some puzzles, since it was derived from that Sudoku Puzzle.

2. Another strength is letting the user create their own validation if they need it, as since this bean is flexible it can be used for various programs so with a set validation scheme it may affected how certain programs worked if they implemented it.

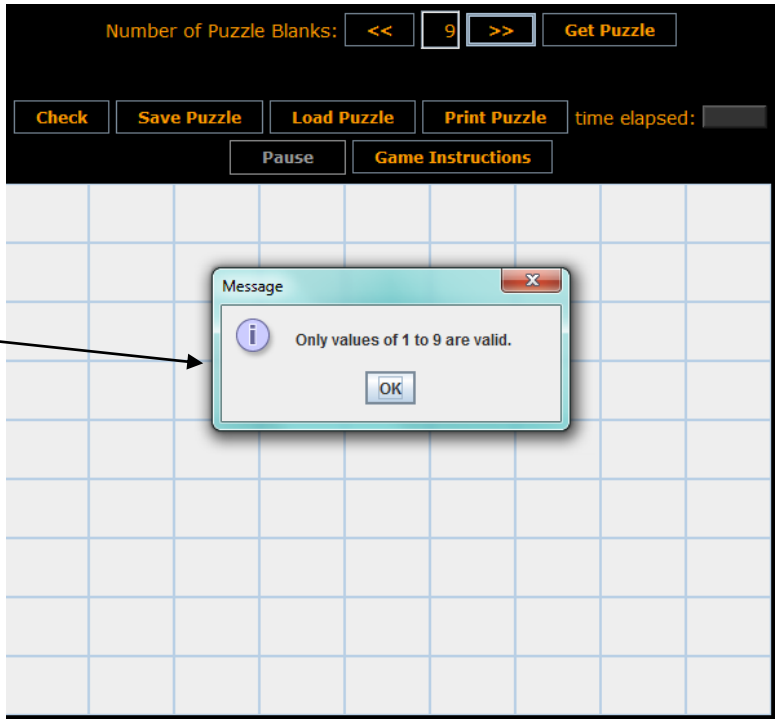
7 – Screenshots demonstrating the completed stages of Part 2

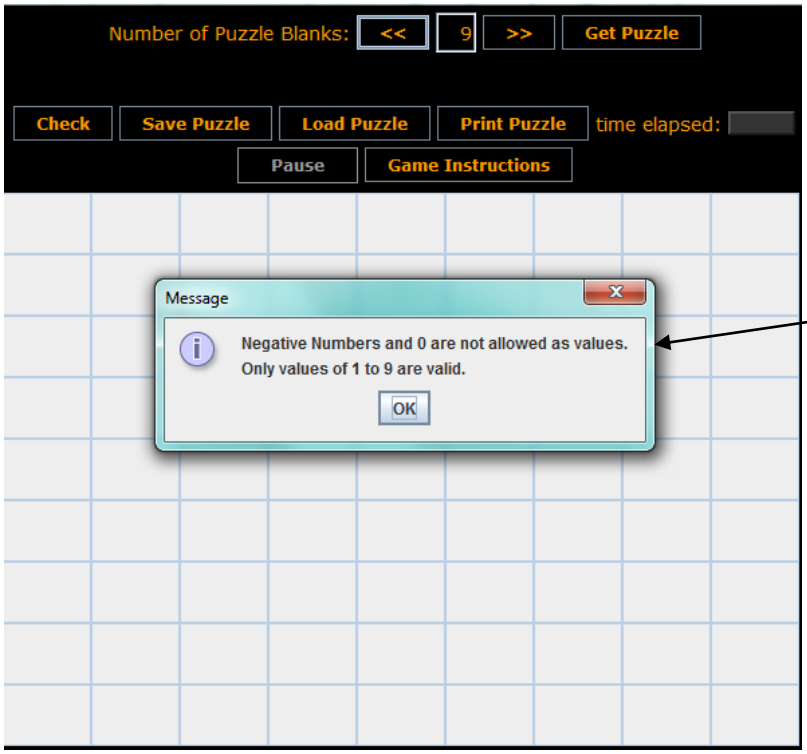
Part 2, Stage 1

Number of Blanks Validation

The screenshot to the right shows the validation added to the program when you try to add more than 9 puzzle blanks to a puzzle.

As you can see a message appears on screen saying that only values of 1 to 9 are valid values.



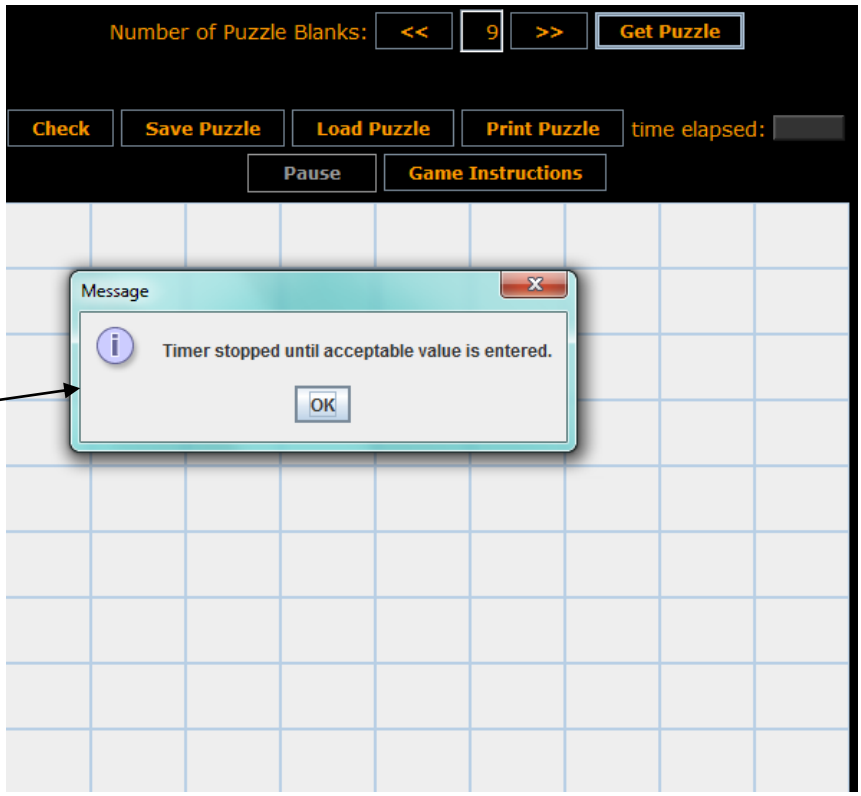


The screenshot to the left shows the other validation message that was placed there, this message appears when the puzzle blanks is set to lower than 1.

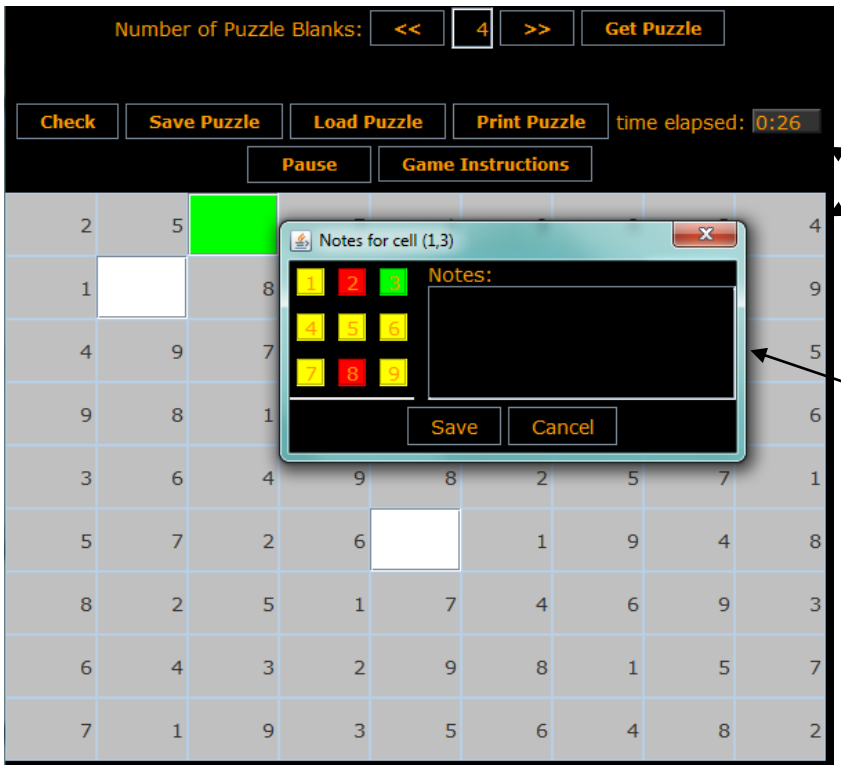
The message indicates that negative numbers and 0 are not valid values and that only 1 to 9 are valid values.

The screenshot to the right shows the validation message stopping the timer and such until an acceptable value is entered.

There is a bug concerning this validation (see: #3 in Bug and Weaknesses section).

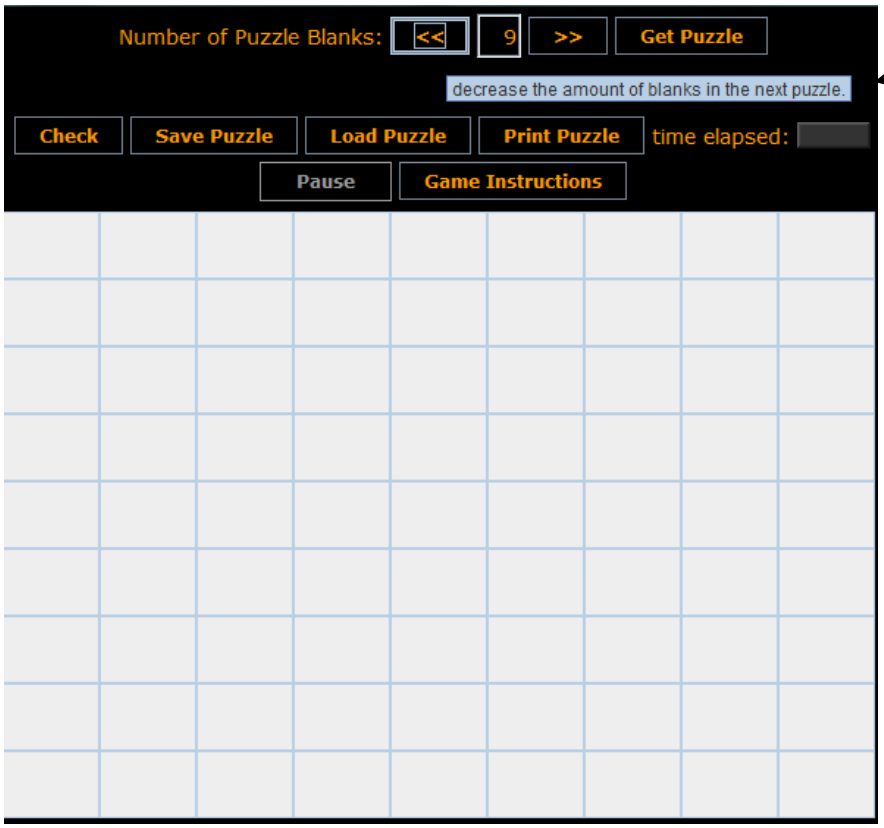


Improving the Overall Look and Feel of the Program



The image to the left shows the improvements made to the overall look and feel of the program. As you can see the Sudoku Puzzle itself is in the background with an orange and black colour scheme.

The Notes View is in the centre of the screen with a similar look to it minus the numbered squares.



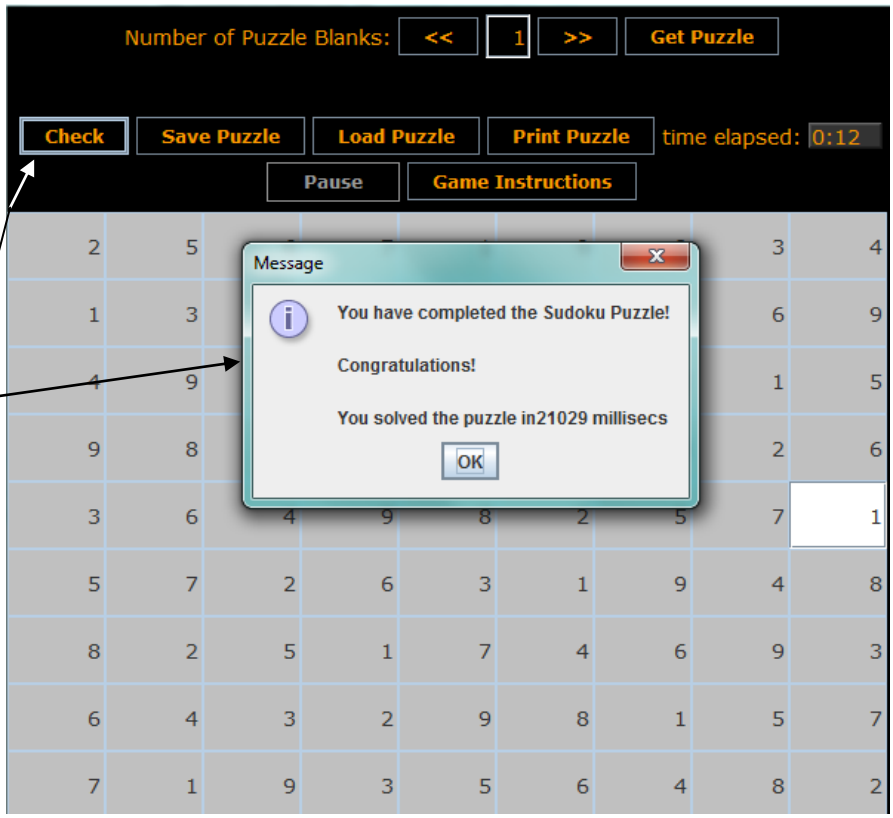
The image to the left shows one of the tooltips I added to the original Sudoku Puzzle.

This one is for the decrease button and indicates that it decreases the amount of blanks in the next puzzle to appear (which is after Get Puzzle has been clicked).

Puzzle Completion Message

The screenshot to the right shows the Puzzle Completion Message which shows up when the user enters the correct number and then clicks the Check button.

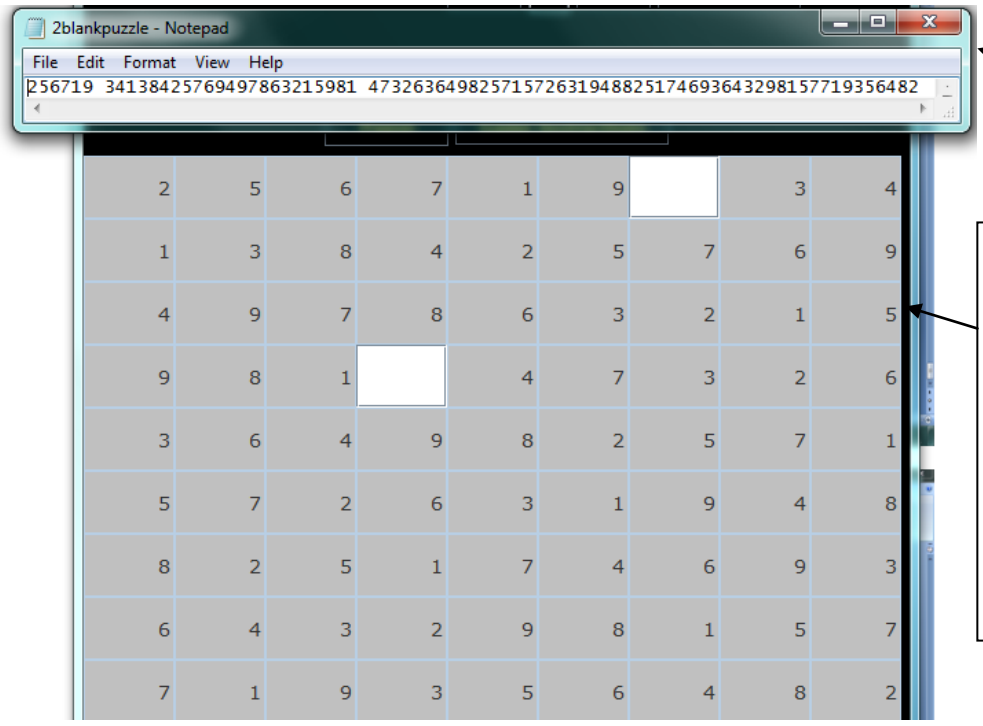
As you can see it indicates that the puzzle was completed correctly, congratulates them and provides the amount of time it took for them to complete it.



Saving to Text File

The screenshot to the right shows the program I generated in the Sudoku Puzzle which shows that it only has two blanks in it.

I then clicked on the Save Puzzle button, named the file and selected where to save it ..

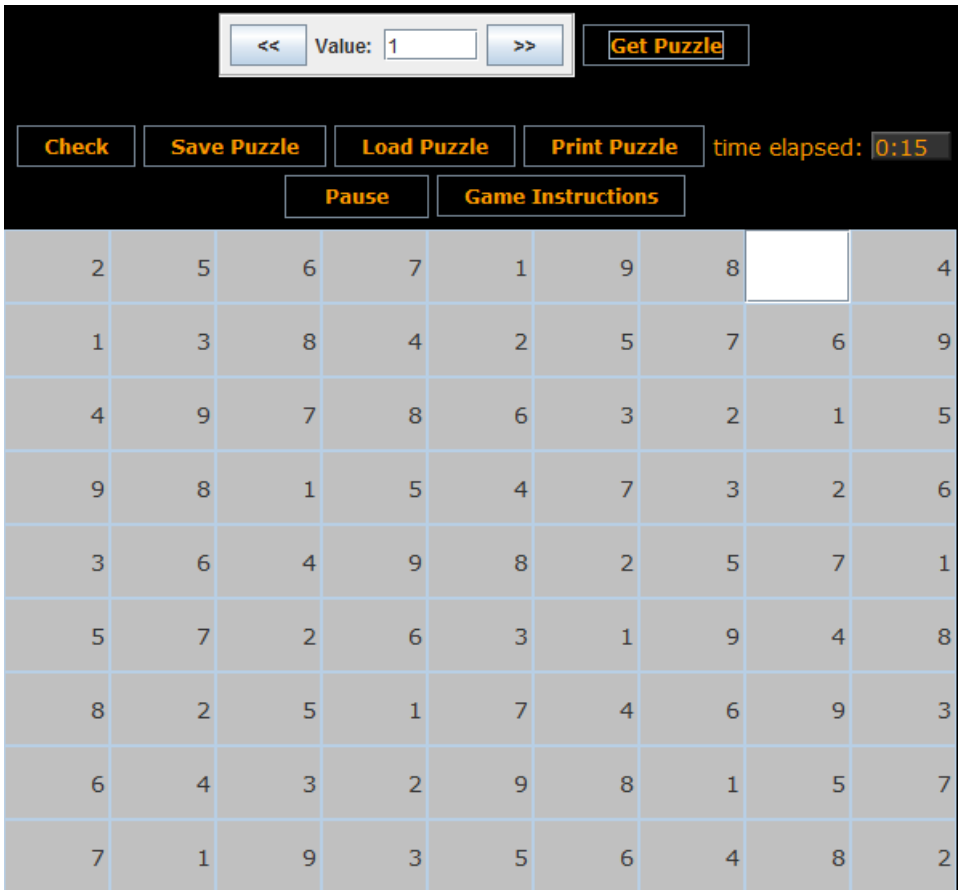


.. when I opened the file this is how it was saved.

I left the original program in the background of the screenshot so both the puzzle and the text file can be compared.

Part 2 Stage 2

Java Bean Sudoku Puzzle

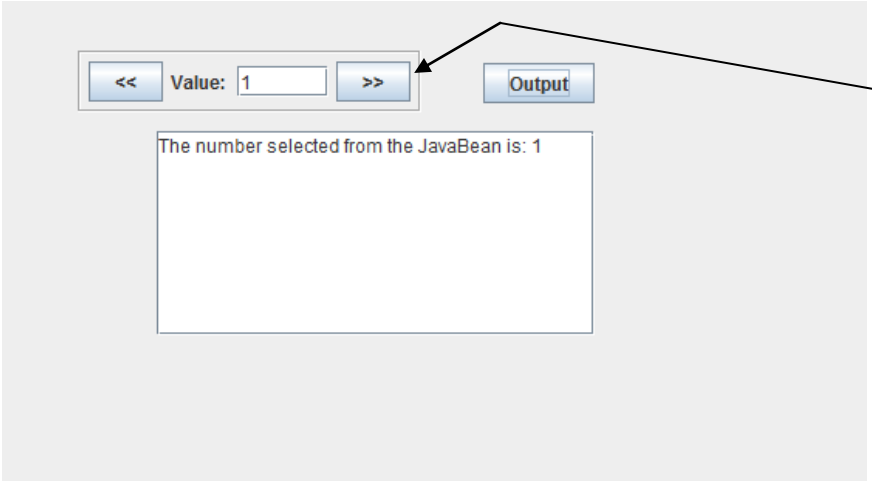


The screenshot the right shows the Java Bean version of the Sudoku puzzle.

As you can see, my Java Bean is there and is capable of generating a puzzle with the value.

Part 2, Stage 3

Other Application to Test Java Bean



The screenshot to the left is the trivial application I used to test my Java Bean in order to see if it could be loaded into NetBeans and incorporated.

As you can see to the left it can be loaded by NetBeans and incorporated into another program.