Contents:

We studied a code for implementing Sudoku for the major portion of the lecture. In the last 10 minutes, we also looked at how we can call one constructor from another using `this ()`, the use of `break` and `continue` statements.

Class Sudoku

This class simulates the Sudoku board. Its important methods are `setCell`, `doRow`, `doColumn` and `doBlock`.

Attributes

BLKSIZE = 3

Block size - Each block is made up of 3×3 cells.

SIZE

The Sudoku board is a square of blocks (defined above). Thus, if $BLKSIZE = 3$ then, a Sudoku board is a square of 3×3 blocks, or 9×9 cells.

board

The Sudoku game board.

Methods

Empty constructor

setCell (int i, int j, String val)

This method sets the specified cell to the value in String val.

```
if ((i<0) || (i >= SIZE) || (j < 0) || (j > SIZE)) return;
```

Do nothing if the indices are invalid.

```
if ((val.length() == 0) || (val.length() > 1))
{
 board[i][j] = new SudokuCell();
 return;
}
if ((val.charAt(0) < '0') || (val.charAt(0) > ('0' + Sudoku.SIZE)))
{
```

```
 board[i][j] = new SudokuCell();
 return;
}
```

`val` is of type String. Since it should contain a single digit signifying the value at the cell, it should be of length exactly equal to 1. This is checked by the first `if` condition. Now we know that the length of the string `val` is 1. But this should also be a digit. That is checked by the next `if` condition. So, now we know that `val` has a single *digit* at index 0.

```
board[i][j] = new SudokuCell(val.charAt(0) - '0');
```

The cell is set with the integer value of `val`. The constructor of `SudokuCell` takes an integer as input. If we had not performed the above checks, then this line (and hence the method) would not have worked as expected by the caller. Please note that 0 is not a valid input. But this line will be executed if `val = "0"` as well. However, this situation is taken care of in the constructor of `SudokuCell`.

doRow(int i)

If a value is frozen for any cell in the ith row, then this method removes this value from the `contents[]` array of all the cells in that row. If the `contents[]` array of atleast one cell is modified in this manner, then the method returns `true`. Else, it returns `false`.

```
if ((i < 0) || (i >= SIZE)) return change;
```

`change` has been set to `false`. Thus, if an invalid row number is given, then the method returns `false`, saying that no changes have been made in any cell.

```
for (j=0; j<SIZE; j++) {
 t = board[i][j].getValue();
```

j holds the column number. `getValue` will return the value at location $[i][j]$ if the value there has been frozen. Else it returns `0`.

```
if (t != 0) {
 int k;
 for (k=0; k<SIZE; k++) {
  if (k != j)
   change |= board[i][k].block(t);
 }
}
```

If the value at $[i][j]$ has been frozen to t, then we proceed to block the value t from all the cells in that row. That is, the `contents[]` array of all the cells in that row (except the one in which the value was found to be frozen, j) will not contain t after this.

If the `contents[]` array of any cell has been modified in this manner, then the boolean indicator, `change` becomes `true`.

doColumn(int i)

The functionality of this method is the same as that of `doRow`. Now we check the cells in each row of the ith column. Hence, we have, `t = board[j][i].getValue();` and `change |= board[k][i].block(t);`. That is, we read the frozen value in each cell in the ith column using `board[j][i].getValue()`. Once we find that a value has been frozen, we check eack cell in the ith row using `board[k][i].block(t)`.

doBlock(int xblk, int yblk)

A *block* is defined to be a square of $BLKSIZE \times BLKSIZE$ cells. And, $BLKSIZE \times BLKSIZE$ such blocks make up a Sudoku board. While playing Sudoku, we also need to check that if a value is frozen in any cell of the block, then it does not appear in the `contents[]` array of any other cell in that block.

The functionality of this method is the same as that of `doRow` and `doCloumn`. That is, if a value is frozen then it must be ensured that it does not appear in any other cell in the block. If a cell's `contents[]` array has been modified, return `true`, else return `false`.

However, this method is the trickiest of all other methods in the class. Reason: The cells are numbered according to the row number and column number in the whole of Sudoku board, and not according to their locations in the block.

The argument for the method specifies which block we should be checking. There are $BLKSIZE$ number of rows and $BLKSIZE$ number of columns. e.g, if $BLKSIZE = 3$, then there are 3 rows and 3 columns of blocks.

```
if ((xblk<0) || (xblk>=BLKSIZE) || (yblk<0) || (yblk>=BLKSIZE))
return change;
```

`change` is initialized to `false`. Hence, if invalid block number is passed as argument, then we return false.

```
for (i=0; i<BLKSIZE; i++)
 for (j=0; j<BLKSIZE; j++)
```

Check for each row and each column in the block.

`t = board[xblk*BLKSIZE + i][yblk*BLKSIZE + j].getValue();` i holds the row number in the block and ranges from 0 to $BLKSIZE - 1$. j holds the column number in the block and ranges from 0 to $BLKSIZE - 1$. We

now need to calculate the absolute row and column numbers (in the Sudoku board).

Please observe that the 0th row, 0th column cell in block $[xblk, yblk]$ could be accessed in the Sudoku board using the index $[xblk * BLKSIZE, yblk * BLKSIZE]$. When the row index or column index in the block increases by 1, the index in the board also increases by exactly 1. Thus, the ith row, jth column cell in block $[xblk, yblk]$ could be accessed in the Sudoku board using the index $[xblk * BLKSIZE + i, yblk * BLKSIZE + j]$. Method `getValue` will return the value at specified location if the value there has been frozen. Else it returns `0`.

```
if (t != 0) {
 int k, l;
 for (k=0; k<BLKSIZE; k++)
  for (l=0; l<BLKSIZE; l++)
   if ((k!=i) || (l!=j))
    change |= board[xblk*BLKSIZE + k][yblk*BLKSIZE + l].block(t);
}
```

If the value at $[i][j]$ in the specified block has been frozen to t, then we proceed to block the value t from all the cells in that block. That is, the `contents[]` array of all the cells in that block (except the one in which the value was found to be frozen, $[i, j]$) will not contain t after this.

If the `contents[]` array of any cell has been modified in this manner, then the boolean indicator, `change` becomes `true`.

Since we need to check cells in various rows and columns when we are checking a block, we need two `for` loops, one for rows and one for columns. Here, k is used to specify the row in the block and l is used to specify the column in the block.

public void Display(SudokuDisplay sd)

This method is used to display the board. It calls a method in the class `SudokuDisplay`. We dont need to concern ourselves with the implementation of this class.

public String toString()

This method is used to convert the Sudoku board to a string.

Class test

This class contains the `main` method.

Methods

static void pause(Scanner s)

This method is called from `main`. It ensures that the user can see the output of each iteration of `doRow`, `doColumn` or `doBlock`. After each call to one of these methods, the user has to press the return key. Only then will the `main` method proceed.

main method

The objects of various classes are initialized. When the object `sd` of class `SudokuDisplay` is created, the sudoku board panel gets displayed since that is the functionality of the constructor of class `SudokuDisplay`. The user enters the puzzle in this panel. `pause(sc);` is used, so that once the user has entered the puzzle, he presses the return key. This signifies to the program that it should start workingon the data. `for (i=0; i<Sudoku.SIZE; i++)`

```
for (j=0; j<Sudoku.SIZE; j++)
 s.setCell(i, j, sd.getValue(i, j));
s.Display(sd);
```

The values entered by the user are given to the respective cell.

```
do {
 change = false;
–
–
–
}while (change);
```

Continue performing the operations until the program is making progress, ie, atleast one change in the `contents[]` array of atleast one cell occurs in each loop.

```
for (i=0; i<Sudoku.SIZE; i++) {
 pause(sc);
 change |= s.doRow(i);
 s.Display(sd);
 System.out.println(‘‘After Row ’’ + i + ‘‘/n’’);
}
```

On each of the rows, `doRow` is called. ie, we check if any cell in that row has been frozen, and change the `contents[]` array of the other cells of that row.

The same is done by considering each column at a time, and each block at a time.

Thus, in the main method, we keep modifying the `contents[]` array of cells in each row, each column and the each block until, no more modifications can be made based on the current data.

– End of program

Calling one constructor from another

Another constructor of the same class can be called using the keyword `this`. E.g.

```
if (val > SIZE)
 this ();
else
 ...
```

Here, the constructor without any arguments has been called.

Statement to change the flow of execution: break

There are statements which change the flow of execution. We have already seen the use of `return` statement to return from any method. Let us now look at another statement: `break`. You can use a `break` statement to terminate a for, while, or do-while loop, as shown in the following example:

```
Scanner sc = ...
while (true){
 int i = sc.nextInt ();
 if (i < 0 || i > 20)
  break;
 System.out.println (fact (i));
}
```

Assume that the method which returns the factorial, `fact (i)` is already written. Let us call this code 1. Now consider Code 2 as follows:

```
boolean done = false; while !done { int i = sc.nextInt (); if
(i <0 || i > 20) done = true; else  System.out.println (fact (i));
}
```

The control follows a straight flow in code 2. This makes it easier to write test cases and debug. Hence, it is difficult to make errors here.

In code 1, the `break` statement makes it a bit more difficult to understand the flow of control. Hence, it is more difficult to write test cases for code 2.

However, in code 1, the compiler does not have to write any statements to check if the condition of `while (true)` statement is true. But, in code 2,

the condition `while (!done)` needs to be checked everytime.

We can thus say that code 1 is more *efficient*, while code 2 is more *user-friendly*.