Relativity
(classical computer programming)

Sudoku rule report
application.
Project 5
Requirements Specification
Java application program
Reads and writes from/to the keyboard/screen
User view of behavior must be the following actions in the sequence written:
1. Read 81 ints (no error checking required)
2. Reprint them in Sudoku board form
3. Check how many times every number 1-9 appears in every row, every column and every block. Print the EXACTLY SPECIFIED MESSAGE in EVERY case of a number either not appearing or appearing more than once.
4. Print the one that's true true two EXACTLY specified messages about the inputs.
Requirements like these can be translated into application code, when you EXPRESS BIG THINGS with METHOD CALLS.

```java
import java.util.Scanner;
public class Pro5App
{
    public static void main(String[] a)
    {
        FillableSudoku mS = new FillableSudoku();
        Scanner sc = new Scanner(System.in);
        mS.read(sc); //Does step 1.
        System.out.println(mS); //Does step 2.
        mS.printReport(); //Does steps 3 and 4.
    }
}
```

YOUR JOB: WRITE, TEST, DEBUG the `printReport()` method plus the 3 helper methods to help `printReport()`.
Proj5 suggested SOFTWARE DESIGN

Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

printReport() (when called) runs though ALL COMBINATIONS of rows, columns or blocks COMBINED WITH ALL NUMBERS 1,2,...,9 and CALLS a helper method for each combination.

printReport() prints the right message EVERY TIME a helper returns a value that is ==0 or > 1. (Requirement 3)

printReport() “remembers and tracks stuff” and prints the right last message. (Requirement 4)
Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

Wise question ??:
Is it a row, a column or a block?

Suggested answer: WHICH OF 3 helper methods, with different method NAMES, answers that question.
Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

?: WHICH row, column or block? (of 9 possibilities!)

Suggested answer: One parameter for LOCATING one row or column, TWO parameters for LOCATIONG one block.
Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

?: WHICH is the given number? (of 9 possibilities!)

Suggested answer: Another parameter.
Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

?): HOW does a parameter rParam locate a ROW?

Suggested answers A: People way—row number 1, 2, ..., 9, so locate a row ELEMENT by

```
this.board[index][ rParam-1 ]
```

B: Geeky way—row index 0, 1, ..., 8, so locate a row elt by

```
this.board[index][ rParam ]
```
Each helper method (when called) loops through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It returns that count.

¿: How do two parameters \( rP \) and \( cP \) locate a block?

Suggested answers:
A: People way—1, 2, 3 in \( rP \); 1, 2, 3 in \( cP \).
B: Geek way—0, 1, 2 etc.
C: Ugly Geek way:
0, 3, 6 in \( rP \); 0, 3, 6 in \( cP \).
(Makes it tough to program printReport() :( That's why it's ugly.)
Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

?: Suppose you choose the People way 1,2,3 and 1,2,3. HOW do two parameters \( r_P \) and \( c_P \) locate THE UPPER LEFT ELEMENT OF THE BLOCK?

Answer:
\[
\text{this.board}[(c_P - 1) * 3][r_P - 1] * 3]
\]
Each helper method (when called) loops through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It returns that count.

?: OK. Once the upper left element is located, how do you locate the other block elements?

Answer use loop control and index variables cIndex and rIndex ranging from 0 to 2, and relativity:

\[
\text{this.board}[(cP - 1)*3 + cIndex] \\
[ (rP - 1)*3 + rIndex]
\]
Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and **counts how many times a given number appears** among those 9 elements. It RETURNS that count.

?: How do you count, and return the count?

```java
int count;//Set up variable to hold current count
count = 0;  //Initialize it to 0.
loop setup and control
{
    if( LOCATED ELEMENT == aGivenNum)
    {
        count = count + 1;
    }
}
return count;
```
iClicker: What happens when

\( \text{LOCATED ELEMENT} == \text{aGivenNum} \)

is false?

A) The loop body is always repeated again.

B) \( \text{count} = \text{count} + 1 \)

C) Nothing to count. The loop might or might not stop.

D) Nothing to count. The loop always stops right away.

E) The method call returns.
Proj5 suggested SOFTWARE DESIGN

Each helper method (when called) LOOPS through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

printReport() (when called) runs though ALL COMBINATIONS of rows, columns or blocks COMBINED WITH ALL NUMBERS 1,2,...,9 and CALLS a helper method for each combination.

printReport() prints the right message EVERY TIME a helper returns a value that is ==0 or > 1. (Requirement 3)

printReport() “remembers and tracks stuff” and prints the right last message. (Requirement 4)
Proj5 suggested SOFTWARE DESIGN

printReport( ) (when called) runs though ALL COMBINATIONS of rows, columns or blocks COMBINED WITH ALL NUMBERS 1,2,...,9 and CALLS a helper method for each combination.

??: HOW
Choice A: (1) Outer loop makes a variable take on 1, 2, 3, ..., 9 to count.
(2) Its body has inner loops to run through all 9 rows, and all 9 columns and all 9 blocks.

Choice B: (1) One or more outer loops make a rSelect and cSelect variable's value select all rows, then all columns, then all blocks.
(2) Its or their bodies have inner loops to make a variable take on numbers 1, 2, 3, ..., 9 to count.
Proj5 suggested SOFTWARE DESIGN

Each helper method (when called) LOOPs through the 9 elements in a row, column or block, and counts how many times a given number appears among those 9 elements. It RETURNS that count.

printReport() (when called) runs though ALL COMBINATIONS of rows, columns or blocks COMBINED WITH ALL NUMBERS 1,2,...,9 and CALLS a helper method for each combination.

printReport() prints the right message EVERY TIME a helper returns a value that is ==0 or > 1. (Requirement 3)

EASY—USE an if statement.
printReport() “remembers and tracks stuff” and prints the right last message. (Requirement 4)

Unanimous voting pattern:
boolean allRulesFollowed;
allRulesFollowed = true;
LOOP CONTROL
{
    if( CURRENT RULE (selected by loop control) is VIOLATED )
    {
        allRulesFollowed = false;
    }
}
if( allRulesFollowed ) { Good news; } else { Bad news; }
Sequential processing you have programmed into your applications' main methods:

Making (instantiating) objects:

```java
EditablePicture bigP = new EditablePicture(
   FileChooser.PickAFile()
);

Picture littleP = new Picture(
   FileChooser.PickAFile()
);
```
Sequential processing you have programmed into your applications' main methods:

Calling methods on those objects:

```java
bigP.copyIn( 20, 100, littleP);
//Purpose: Make the code defined in copyIn copy the little Picture into the big Picture.
bigP.explore();
//Purpose: Make you see the result of the copyIn operation.
```
iClicker: Interchange 2 lines of code. What would happen?

main(...) { ...
    // as before, make bigP and littleP ONLY.
    bigP.explore();
    bigP.copyIn(20, 100, litP);
}

(A) You see nothing.
(B) You see the original big picture.
(C) You see the big picture with the little picture copied into it.
(D) You see the little picture.
(E) You see the big picture twice: 1. The original
    2. Original with the little picture copied into it.
The programs below do **different things**!

```java
main(..) { ...//as before
    //make bigP and littleP ONLY.
    bigP.explore();
    bigP.copyIn( 20, 100, litP);
}
```

```java
main(..) { ...//as before
    //make bigP and littleP ONLY.
    bigP.copyIn( 20, 100, litP);
    bigP.explore();
}
```
Which version can you use to TEST your Lab8 copyIn method?

main(..) { ...//as before
    //make bigP and littleP ONLY.
    bigP.explore();
    bigP.copyIn( 20, 100, litP);
}

main(..) { ...//as before
    //make bigP and littleP ONLY.
    bigP.copyIn( 20, 100, litP);
    bigP.explore();
}
DEFINING METHODS

Sequential processing you have programmed into METHOD DEFINITIONS that EXTEND 10-year old preprogrammed behaviors

class EditablePicture extends Picture
{
    public void copyIn(int originX,
                        int originY,
                        Picture source)
    {
        // You coded in Lab8
        return;  // Just return to caller
        // with no return value.
    }
}
Methods that MODIFY, COPY IN or COPY OUT image information of Pixels in Pictures

DEFINING THEM:
USING parameter variables, LOOPS and CONDITIONALS

CALLING THEM: Using parameter values
Laboratory 6 First Half

Before

After

Second Half

origin x, origin y

width, height
Before

During
Calculations done inside copy in to locate:

- \( \text{origin X} + x_{\text{rel}} \)
- \( \text{origin Y} + y_{\text{rel}} \)
copyIn solution

- copyIn method preprograms doing stuff ON (this) EditablePicture object

- Parameter variables:
  - originX
  - originY
  - source (Picture)

- It uses two return values from
  - source.getWidth()
  - source.getHeight()
public class EditablePicture extends Picture
{
    public void copyIn(int originX, int originY, Picture source)
    {
        for(int yRel=0; yRel<source.getHeight(); yRel=yRel+1)
        {
            for(int xRel=0; xRel<source.getWidth(); xRel=xRel+1)
            {
                Pixel srcP = source.getPixel(xRel,yRel);
                this.getPixel(
                        originX+xRel,
                        originY+yRel )
                    .setColor(
                            srcP.getColor());
            }
        }
    }
}
Other purposes for xRel and yRel are:

(1) To track a Pixel location ACTUALLY IN the little Picture.
(2) To help track and compute a Pixel location in the big Picture RELATIVE TO (originX, originY)
public class EditablePicture extends Picture {
    public void copyIn(int originX, int originY, Picture source) {
        for (int yRel = 0; yRel < source.getHeight(); yRel = yRel + 1) {
            for (int xRel = 0; xRel < source.getWidth(); xRel = xRel + 1) {
                Pixel srcP = source.getPixel(xRel, yRel);
                this.getPixel(
                        originX + xRel,
                        originY + yRel)
                    .setColor(srcP.getColor());
            }
        }
    }
}
Homework practice quiz: In what order (1st, 2nd, 3rd, 4th, 5th) does the computer do these 5 things?

1. Get a Pixel from this, big Picture?
2. Compute originX+xRel and originY+yRel?
3. Get a Color from a Pixel of the source, little Picture?
4. Get a Pixel from the source, little Picture?
5. Set the big Picture's Pixel to a usually different Color?
public class EditablePicture extends Picture
{
    public void copyIn(int originX, int originY, Picture source)
    {
        for(int yRel=0; yRel<source.getHeight(); yRel=yRel+1)
        {
            for(int xRel=0; xRel<source.getWidth(); xRel=xRel+1)
            {
                Pixel srcP = source.getPixel(xRel, yRel);
                this.getPixel(
                        originX+xRel,
                        originY+yRel)
                    .setColor(
                            srcP.getColor());
            }
        }
    }
}
public class EditablePicture extends Picture {
    public void copyIn(int originX, int originY, Picture source) {
        for(int yRel=0; yRel<source.getHeight(); yRel++) {
            for(int xRel=0; xRel<source.getWidth(); xRel++) {
                Pixel srcP = source.getPixel(xRel, yRel);
                this.getPixel(originX+xRel, originY+yRel).setColor(srcP.getColor());
            }
        }
    }
}
A) yRel is used to control a loop.

B) yRel is used to help locate a Pixel within the little source Picture absolutely (not relatively).

C) yRel is used to help locate a Pixel within this big Picture relative to (originX, originY).
public class EditablePicture extends Picture {
    public void copyIn(int originX, int originY, Picture source) {
        for(int yRel=0; yRel<source.getHeight(); yRel=yRel+1) {
            for(int xRel=0; xRel<source.getWidth(); xRel=xRel+1) {
                Pixel srcP = source.getPixel(xRel, yRel);
                this.getPixel(originX+xRel, originY+yRel)
                    .setColor(srcP.getColor());
            }
        }
    }
}
A) yRel is used to control a loop.

B) yRel is used to help locate a Pixel within the little source Picture absolutely (not relatively).

C) yRel is used to help locate a Pixel within this big Picture relative to (originX,originY).
public class EditablePicture extends Picture
{
    public void copyIn(int originX, int originY, Picture source)
    {
        for(int yRel=0; yRel<source.getHeight(); yRel=yRel+1)
        {
            for(int xRel=0; xRel<source.getWidth(); xRel=xRel+1)
            {
                Pixel srcP = source.getPixel(xRel, yRel);
                this.getPixel(originX+xRel, originY+yRel)
                    .setColor(srcP.getColor());
            }
        }
    }
}
A) $y_{Rel}$ is used to control a loop.

B) $y_{Rel}$ is used to help locate a Pixel within the little source Picture absolutely (not relatively).

C) $y_{Rel}$ is used to help locate a Pixel within this big Picture relative to $(\text{originX,originY})$. 
Purposes for xRel and yRel are:
(1) To track an ABSOLUTE Pixel location ACTUALLY IN the little Picture.

(2) To help track and compute a Pixel location in the big Picture RELATIVE TO (originX, originY)
Calculations done inside copy in to locate
origin \( X + x_{rel} \)
origin \( Y + y_{rel} \)
You can program anything you want (like in Alice's restaurant)

I programmed copyOut to STOP about 1/3 through its original task.

HOW? I made another variable for COUNTING each Pixel when it is re-colored. Also code to compute HOW MANY Pixels in the little Picture.
public void copyIn(int originX, int originY, Picture source)
{
    int nPixCopied = 0;
    for(int yRel=0; yRel<source.getHeight(); yRel=yRel+1)
    {
        for(int xRel=0; xRel<source.getWidth(); xRel=xRel+1)
        {
            Pixel srcP = source.getPixel(xRel,yRel);
            this.getPixel(
                originX+xRel,
                originY+yRel).setColor(srcP.getColor());
            nPixCopied=nPixCopied+1;
            if(    nPixCopied
                  ==
                  (int)(0.3251*source.getHeight()*source.getWidth())  )
            {
                return;
            }
        }
    }
}
About 0.3 of the full rows plus one partial row.
Sudoku
too hard for now!

Sudoku BOARD computing support
CURRENT LAB 10!
Sudoku board

- 81 squares or boxes for the maker and puzzle solver to write in numbers 1,2,...9.
- 9 rows of 9 boxes each. In other words, 9 columns of 9 boxes each. Just like a digital Picture.
- 3 rows of 3 BLOCKS each---A BLOCK is a 3x3 grid of boxes!
- In other words, 3 columns of 3 BLOCKS each. (Confusing!)
3 kinds of Sudoku rules (too hard for now)

- **Row rule:** Each of the 9 boxes in a row must contain exactly one of each of the 1-9 numbers.
- **Column rule:** Each of the 9 boxes in a column must contain exactly one of each of the 1-9 numbers.
- **BLOCK rule:** Each of the 9 boxes in a (3x3) block must contain exactly one of each of the 1-9 numbers.

There are 27 rules altogether: 9 different row rules, 9 different column rules and 9 different block rules!
The lecture now and this week's lab10 is about LOCATING board boxes or squares within rows within columns within blocks
READ
THE
JAVADOC

Javadoc web site for Sudoku and other classes
Live coding example 1: Write a 7 into EACH of the 9 upper left boxes of each of the 9 BLOCKS

```
7 0 0 7 0 0 7 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

7 0 0 7 0 0 7 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

7 0 0 7 0 0 7 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```
public void sevensInULsOfBlocks()
{
    for( int rb = 0; rb < 3; rb = rb + 1)
    {
        for( int cb = 0; cb < 3; cb = cb + 1)
        {
            this.board[3*cb][3*rb] = 7;
        }
    }
    return ;
}
Live coding example 2: Write 1-9 into EACH of the 9 upper left boxes of each of the 9 BLOCKS

1 0 0 2 0 0 3 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

4 0 0 5 0 0 6 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

7 0 0 8 0 0 9 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
public void countsInULsOfBlocks()
{
    int blkCounter = 1;
    for( int rb = 0; rb < 3; rb = rb + 1)
    {
        for( int cb = 0; cb < 3; cb = cb + 1)
        {
            this.board[3*cb][3*rb] = blkCounter;
            blkCounter = blkCounter + 1;
        }
    }
    return ;
}
Live coding example 3: Write 1-9 into EACH of the 9 second-row, third-column boxes RELATIVE TO EACH of the 9 BLOCKS

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
public void countsIn23RelativesOfBlocks()
{
    int blkCounter = 1;
    for( int rb = 0; rb < 3; rb = rb + 1)
    {
        for( int cb = 0; cb < 3; cb = cb + 1)
        {
            this.board[3*cb+2][3*rb+1] = blkCounter;
            blkCounter = blkCounter + 1;
        }
    }
    return ;
}
Back to copyIn
I programmed copyOut to TEST if (originX+yRel,originY+yRel) is a valid Pixel array location BEFORE trying to get a Pixel from a location where there might be NO Pixel. Purpose: Avoid a CRASH.

(You can program anything you want like in Alice's restaurant.)
This photo of Theresa's Stockbridge Cafe is courtesy of TripAdvisor
public void copyIn(int originX, int originY, Picture source)
{
    for(int yRel=0; yRel<source.getHeight(); yRel=yRel+1) {
        for(int xRel=0; xRel<source.getWidth(); xRel=xRel+1) {
            Pixel srcP = source.getPixel(xRel, yRel);
            if( originX+xRel < this.getWidth()
                &&
                originY+yRel < this.getHeight() ) {
                this.getPixel(
                        originX+xRel,
                        originY+yRel).setColor(srcP.getColor());
            }
        }
    }
}
Result

The copied in (small) Picture is “cropped” on the left and/or on the bottom if it would otherwise go outside this (big) Picture.

It led me to think of an INNOVATION!

The SAME if conditional statement technique can be used to crop the little Picture on the right and/or top also!
public void copyIn(int originX, int originY, Picture source) {
    for (int yRel = 0; yRel < source.getHeight(); yRel = yRel + 1) {
        for (int xRel = 0; xRel < source.getWidth(); xRel = xRel + 1) {
            Pixel srcP = source.getPixel(xRel, yRel);
            if (originX + xRel < this.getWidth() && originX + xRel >= 0 && originY + yRel < this.getHeight() && originY + yRel >= 0) {
                this.getPixel(originX + xRel, originY + yRel).setColor(srcP.getColor());
            }
        }
    }
}