

- Let's ask something different. Let's exclude the bit where I selected blocks. But can you explain my mental process to code where I began filling the board. I am not after explanation of validation if duplicate values. I am more interested in on how I populated numbers and offset logic

Thought for 20s >

Love this question — this is the fun part: reverse-engineering how you were thinking when you wrote it 😊

I'll ignore the random 3×3 selection and the row/column validation, and just walk through: how

- you mapped a 3×3 (temp) into the 9×9 (nineByNine)
- what you were trying to do with rowIndex , colIndex , rowCount , colCount and offset
- what the boolean flags and counters were doing.

1. Your mental model in one sentence

You essentially thought:

“I'll stream 81 numbers into the board one at a time, but I want to think in terms of 3×3 blocks, so I'll track:

- “where I am inside the current 3×3 (rowCount , colCount), and”
- “where I am inside the full 9×9 (rowIndex , colIndex),
and use offset + counters to move each 3×3 block to the right place.””

So you didn't compute the block position with a formula like $\text{startRow} = (\text{block} / 3) * 3$; instead you maintained a kind of “cursor” that walked around the board while you were filling blocks.

2. The two coordinate systems you used

You had two separate coordinate spaces:

- **A. Inside the current 3×3 grid (temp)**

java

 Copy code

```
int temp[][] = new int[3][3];
```

```
int rowCount = 0; // which row in temp (0..2)
int colCount = 0; //
which col in temp (0..2)
```

This is the local position of the number within the small 3×3:

- (rowCount, colCount) goes from (0,0) through (2,2) .
- After 9 steps → you’ve processed the whole mini-grid.

B. Inside the 9×9 Sudoku (nineByNine)

java

 Copy code

```
int rowIndex = 0; // which row in the 9x9
int colIndex = 0; // which col in the 9x9
int offset;      // starting column of the *current* 3x3 block in this band
```

This is the global position on the final board.

You constantly did:

java

 Copy code

```
nineByNine[rowIndex][colIndex] = temp[rowCount][colCount];
```

So mentally you were always asking:

“Given where I am in the mini-grid (rowCount , colCount), where should that go in the

```
big board ( rowIndex , colIndex )?"
```

3. The big picture of how you wanted to fill the board

You wanted to fill the 9×9 as:

- 3 blocks per row band, and 3
- row bands:

```
text
```

[Copy code](#)

```
[block1][block2][block3]
[block4][block5][block6]
[block7][block8][block9]
```

Each block is 3×3. So globally a block position is:

- block 1 → rows 0–2, cols 0–2
 - block 2 → rows 0–2, cols 3–5
 - block 3 → rows 0–2, cols 6–8
 - block 4 → rows 3–5, cols 0–2 etc.
 - Instead of computing that directly with formulas, you used:
 - `totalNumbersProcessed` to figure out which band you're in,
 - `i / numberOf3x3Processed` to figure out which block you're in,
 - `offset` to remember where the block started horizontally.
-

4. How `totalNumbersProcessed` & the three conditions worked

You had this bit:

```
java
```

[Copy code](#)

```
if (totalNumbersProcessed<=27 && !condition2 && !condition3)
{
    rowIndex=0;    colIndex=0;
    condition1=true;
    condition2=true;
    condition3=true; }
```

```

if (totalNumbersProcessed<=54 && totalNumbersProcessed>27 && condition1 && conditi
{
    rowIndex=3;    colIndex=0;
condition2=true;
condition1=false;
condition3=false;
}

if (totalNumbersProcessed<=81 && totalNumbersProcessed>54 && condition2 && !condit
{
    rowIndex=6;    colIndex=0;
condition3=false;
condition1=true;
condition2=false;
}

```

What you were thinking totalNumbersProcessed runs

- from 1 to 81.
- You split this into three chunks of 27 numbers:
 - 1–27 → first band (rows 0,1,2), 28–54
 - → second band (rows 3,4,5), 55–81 →
 - third band (rows 6,7,8).
- The conditions condition1/2/3 are just “locks” to make sure each band’s starting point is set only once.

So mentally:

- When you start (first cell), you set rowIndex = 0 and colIndex = 0 → top-left of the board.
- When you hit cell 28, you set rowIndex = 3 , colIndex = 0 → start of middle band.
- When you hit cell 55, you set rowIndex = 6 , colIndex = 0 → start of bottom band.

That ensured the vertical placement of blocks: first 3 blocks in rows 0–2, next 3 in rows 3– 5, last 3 in rows 6–8.

5. How you moved inside a 3×3 and across rows: rowCount ,
colCount ,
rowIndex++

You had:

java

 Copy code

```
for (int n=0; n<temp.length; n++) // rows inside temp (0..2)
{ if (colCount>=2 && rowCount!=2)
{
    rowCount++; // next row in temp    colCount=0; // back to
first column in temp    rowIndex++; // move down one row in the
9x9

    if (offset==0)
    {
        colIndex=0;
    } else {
        colIndex=offset;
    }

    if (colIndex==8 && numberOf3x3Processed==9)
    {
        colIndex=0;    rowIndex=rowIndex+3;
    }

    if (colIndex!=8 && numberOf3x3Processed==9)
    {
        colIndex=colIndex+1;    rowIndex=0;
    }
}

for (int k=0; k<temp[0].length;k++)
{
    numberOf3x3Processed++;

    if (k==0)
    {
        offset=colIndex; // store starting column of this block
    }

    // place the number    nineByNine[rowIndex][colIndex] =
temp[rowCount][colCount];

    // advance inside 3x3 and 9x9
    colCount++;    colIndex++;
}
```

```

        if (numberOf3x3Processed==9)
        {
            colCount=0;        rowCount=0;
        }
    }
}

```

Mentally, what this did `rowCount` / `colCount` walk the 3×3 in

- row-major order.
- `rowIndex` / `colIndex` walk whatever part of the 9×9 you're currently filling.
- When `colCount >= 2` , you reached the end of a row in the 3×3 : You
 - increment `rowCount` (go to next row in temp), reset `colCount` to 0,
 - increment `rowIndex` (go to next global row), reset `colIndex` to the starting column of this
 - block, which is saved in `offset` .

So:

- `offset` is “where this 3×3 started horizontally in the big grid”.
- Every new row of temp uses the same `offset` to continue under the same 3×3 column group.

Example for block 1 (top-left):

- Start: `rowIndex=0` , `colIndex=0` , so `offset=0` .
- First row of temp writes to (0,0), (0,1), (0,2).
- On new row of temp:
 - `rowIndex=1` , `colIndex=0` → second row writes to (1,0),(1,1),(1,2).
- Next row: `rowIndex=2` , `colIndex=0` → (2,0),(2,1),(2,2).

So that local logic correctly fills each block as a 3×3 rectangle in the right columns.

6. How you jumped between blocks horizontally: `offset` and the endof-block logic

Two key bits:

java

 Copy code

```

if (k==0) {
    offset = colIndex;
}

```

This means:

- On the first cell of each row of the 3×3 , you record colIndex as offset .
- That marks: “this block starts at column X in the big grid”.

Then at the end of rows and end of a whole 3×3 block:

java

 Copy code

```
if (colIndex==8 && numberOf3x3Processed==9)
{
    colIndex=0;    rowIndex=rowIndex+3;
}

if (colIndex!=8 && numberOf3x3Processed==9)
{
    colIndex=colIndex+1;    rowIndex=0;
}
```

What you were aiming for

You wanted:

- After finishing block 1 at (rows 0–2, cols 0–2), the next block (block 2) should start at (row 0, col 3).
- After block 2 (rows 0–2, cols 3–5), block 3 starts at (row 0, col 6).
- After block 3 (rows 0–2, cols 6–8), the next block (block 4) should start at (row 3, col 0) — i.e. next band.

So this logic says:

- If we just finished a 3×3 (numberOf3x3Processed==9), and
 - We’re at column 8 → we’re at the end of the band horizontally:
 - reset colIndex to 0, jump rowIndex += 3 → drop down to the next band.
 - We’re not at column 8 → we’re in the middle of a band:
 - colIndex++ → move one column further; that becomes the new block start (so effectively 3 columns over over the course of the next mini-grid filling),
 - rowIndex = 0 → back to the top row of that band.

In other words:

- The “end-of-block” checks tried to hop you between block 1 → 2 → 3, and then down to the next row band 4 → 5 → 6, and so on, without explicitly computing the (blockRow, blockCol) formula.

7. The `i` and `numberOf3x3Processed` switch block

You also had:

java

 Copy code

```
if (numberOf3x3Processed==9)
{ switch(i)
  { case 1: case 2: case 3:
    rowIndex=0;
    break;
    case 4: case 5: case 6:
    rowIndex=3;
    break;
    case 7: case 8: case 9:
    rowIndex=6;
    break;
  }
}
```

Here:

- `i` is the block number (1..9), incremented every time a 3×3 is finished. `numberOf3x3Processed`
- tells you “we’ve just completed a mini-grid”.

Mentally

This was like a backup / safety mechanism you added:

- “If I’m on block 1–3, make sure `rowIndex` is 0”
- “If I’m on block 4–6, make sure `rowIndex` is 3”
- “If I’m on block 7–9, make sure `rowIndex` is 6”

So even if the earlier condition 1/2/3 logic messed up, this switch would try to snap the block back into the correct band.

It’s a bit redundant with the `totalNumbersProcessed` logic, but it shows your mental process:

“I want block 1–3 in band 0, 4–6 in band 1, 7–9 in band 2 — I don’t fully trust the earlier logic, so I’ll enforce it again here.”

8. Putting it all together: your mental algorithm

If I summarise your board-filling thought process:

1. Think in terms of 9 mini 3×3 grids that must be placed in a 3×3 block layout inside the 9×9 .
2. Use a single stream of 81 steps (`totalNumbersProcessed`) to drive everything.
3. Split those 81 steps into three chunks of 27 to decide which row band (0–2, 3–5, 6–8) you're in.
4. Within each mini-grid:
 - `rowCount / colCount` walk a 3×3 , `rowIndex / colIndex` walk the corresponding 3×3 area in the 9×9 .
5. Use offset to “remember” the starting column of the current block, so when you move to the next row of that block, you jump back to that starting column.
6. At the end of a mini-grid (`numberOf3x3Processed==9`), use:
 - `colIndex==8` vs `colIndex!=8` to decide whether to go to:
 - the next block on the right, or the
 - block in the next band down.
 - plus a `switch(i)` as an extra guarantee that block 1–3 stay in band 1, 4–6 in band 2, 7–9 in band 3.

In modern hindsight, you can replace all of that with a couple of clean formulas:

[↩ Copy code java](#)

```
int blockRow = block / 3; // 0,0,0,1,1,1,2,2,2
```

```
int blockCol = block % 3; // 0,1,2,0,1,2,0,1,2
```

```
int startRow = blockRow * 3; int startCol
```

```
= blockCol * 3; rowIndex = startRow +
```

```
rowCount; colIndex = startCol +
```

```
colCount;
```

...but your original logic shows a very natural “let me keep state and move a cursor around” mindset. You were essentially simulating the movement over the grid instead of computing each position from scratch.

If you like, I can rewrite just the population part in your style but with the formula-based row/col calculation — same idea, far less juggling of flags.



So you mentioned simulating movement over the grid..where would my approach be useful in real world

Thought for 8s >

Good question—this is the interesting bit.

When I said you were “simulating movement over the grid,” what you basically did was:

Build a stateful cursor that walks around a 2D grid in a structured pattern (in 3×3 chunks), using `rowIndex` , `colIndex` , `rowCount` , `colCount` , `offset` , and `flags` to control where you go next.

That mental model totally shows up in real-world problems, just not usually with Sudoku level combinatorics bolted on top.

Let me map your approach to some concrete areas.

1. Game development & maps

Think of grid-based games:

- tile maps (2D RPGs, roguelikes), strategy
- games (chess, Go, tactics games), platformers
- with tile-based levels.

Your style of code is very similar to:

- A camera or viewport moving over a larger tile map.
- A cursor that fills parts of the world chunk by chunk.
- Placing prefab “rooms” or “rooms made of 3×3 tiles” into a big dungeon.

Where your approach fits

- You treated the board as 9 blocks of 3×3 and moved a cursor through each block and through the board.

- In games, you might:
 - Load or generate a level block-by-block (e.g., 16×16 chunks),
 - Use an offset to say “this room starts at (x, y) in the world”,
 - Then walk through the room’s local coordinates (localRow, localCol) and map them to (globalRow, globalCol) — exactly what you did with rowCount / colCount \rightarrow rowIndex / colIndex .

That “local \rightarrow global with an offset” pattern is everywhere in level editors and procedural map generation.

2. Image processing & sliding windows

Your “ 3×3 inside 9×9 with offsets” looks a lot like:

- A sliding window or kernel moved over an image:
 - blur filters,
 - edge detection, convolution in
 - neural networks.

Real-world example:

- You have a big 2D array of pixels (say 1024×1024),
- You move a 3×3 or 5×5 window,
- At each position, you read from the window and write a result into an output image.

You did the same kind of mapping:

- A small 3×3 local coordinate system (rowCount , colCount),
- A global offset (rowIndex , colIndex , offset) that says where that window or block sits on the big array,
- And then you stored values into the big structure using those coordinates.

Your Sudoku grid is tiny, but the mental model is exactly the same as:

“Take a block, place it at (startRow, startCol), now walk over that block and write values into the big grid.”

3. Robotics / grid worlds / path planning

Imagine a cleaning robot moving in a grid:

- It has a current cell (rowIndex, colIndex) ,
- It decides movement based on where it is, what it's already cleaned, and sometimes sub-regions (like rooms or zones).

Your approach is close to what you do when you:

- Simulate a robot sweeping a room sector-by-sector,
- Or simulate an agent scanning a 2D grid in “bands” or “chunks”.

The way you used:

- totalNumbersProcessed to decide which band (top/middle/bottom),
- and flags (condition1 , condition2 , condition3) to only set starting rows once, is exactly the

kind of stateful movement logic you'd see in:

- Grid search,
- Coverage planning,
- Simulating something “walking” over a grid in a non-trivial pattern.

4. Layout / tiling / UI grids

Your offset logic (“this 3×3 starts at column 3”, “next block starts at column 6”, etc.) is very similar to:

- Laying out widgets in a grid, Placing
- subpanels into a dashboard, Tiling
- bitmaps into a larger canvas.

Pattern:

1. You have a small component (3×3 block, a panel, a tile).
2. You want it at position (startRow, startCol) in a bigger 2D layout.
3. You iterate local coordinates and compute global positions with:

```
java
```

 Copy code

```
globalRow = startRow + localRow;  
globalCol = startCol + localCol;
```

You did exactly this but drove it via state (offset , rowIndex , colIndex) instead of a direct formula.

That's still a completely valid way to think, especially when:

- Movement depends on previous moves (like your bands and block transitions), Or you're
- modelling something that “walks” rather than just stamping blocks.

5. Streaming / on-the-fly processing

One subtle but important thing you did:

You didn't build the whole board in one go and then validate; you streamed values in and updated the board as you went.

That pattern is used in:

- Real-time data grids,
- Streaming simulations,
- Hardware/embedded systems where you read one element at a time and map it to a 2D structure.

Your `totalNumbersProcessed` , `rowIndex` , `colIndex` effectively defined a stream \rightarrow 2D mapping:

- “Given that this is the k -th number I've seen, where does it go on my 9×9 ?”

That's a mental model that pops up when you:

- Read data from a file/stream/sensor,
 - And need to place it into a grid or paged structure without loading everything at once.
-

6. So what's the core transferable idea?

Underneath all the Sudoku stuff, you built:

1. A cursor that lives on a 2D grid (the 9×9 board).
2. A local view/block (3×3 temp) that you repeatedly map into global positions.
3. A bunch of state variables that control:
 - where each block starts (offset , row bands), how you
 - advance row by row and block by block.

This is useful whenever you have:

- Local structures (tiles, patches, rooms, windows),
- You need to place or move them inside a bigger 2D structure,
- And you want to simulate movement or progression rather than just compute positions formulaically.

If you wrapped that into a little “GridCursor” class, you'd basically have a reusable tool for:

- games,
 - layout engines,
 - image processing,
 - simulations on grids.
-

You always use words like simulator for my code in past also. But explain clearly what aspects make it applicable

Stopped thinking >

You always use words like simulator for my code in past also. But explain clearly what aspects make it applicable

4. You use local coordinates + offsets like a physics / graphics sim

In fill9x9 , for each 3×3 temp , you track:

- Local coordinates inside the block:
 - rowCount , colCount → 0..2
- Global coordinates on the board:
 - rowIndex , colIndex
- Offset for where the block “lives” in the big grid: offset =
 - starting column for this block in its band.

That’s exactly the pattern in many simulations:

- Local coordinates of an object or tile,
- Plus an offset that tells you where it sits in world space,
- Then you map (localRow, localCol) → (globalRow, globalCol) .

Example from your code:

java

 Copy code

```
if (k == 0) {  
    offset = colIndex;      // remember where this block starts horizontally } // ...  
nineByNine[rowIndex][colIndex] = temp[rowCount][colCount];
```

That’s “place this local structure into world coordinates with a moving offset” — same mental model as:

- placing sprites into a game map,
 - placing tiles in a tile engine,
 - or placing elements in a simulation grid.
-

5.



I was mainly interested in point 4 and filling of board

Thought for 9s >

Gotcha – let’s zoom in only on the “fill the board” bit:

how your code maps numbers from the 3×3 temp block into the 9×9 nineByNine using rowIndex , colIndex , rowCount , colCount , and offset .

I’ll ignore validation and random selection and just talk about the movement / offsets.

1. Two coordinate systems: local vs global

2. How offset works (the anchor for a 3×3 block)

Inside the inner loop you have:

```
java
```

```
for (int k = 0; k < temp[0].length; k++) {  
    numberOf3x3Processed++;  
  
    if (k == 0) {  
        offset = colIndex; // remember where this 3x3 starts horizontally    }  
  
    // place    nineByNine[rowIndex][colIndex] = temp[rowCount][colCount];
```

```
colCount++; // move right in the mini-grid    colIndex++; //
move right in the big grid }
```

What you're doing:

- At the start of each row of the 3×3 ($k == 0$), you store the current `colIndex` into `offset` .
- That means:

“This block's left edge, in the 9×9 , is at `column = offset` .”

Later, when you move to the next row of the mini-grid, you use this `offset` to jump the global cursor back to the start of the block in that band:

java

 Copy code

```
if (colCount >= 2 && rowCount != 2) {
    rowCount++;    // next row in temp    colCount = 0;    // back
    to first col in temp    rowIndex++;    // go to next row in 9x9

    if (offset == 0) {
        colIndex = 0;    } else {
        colIndex = offset;    // jump back to block's starting column
    }
}
```

So for each 3×3 block:

- First row: `offset` is set to the starting column in the 9×9 .
- Next row: you go down one row in both `temp` and the big board, then reset `colIndex` back to `offset` , so you're aligned under the same 3-column region.

This is exactly the “anchor + local coordinates” pattern:

```
globalCol = offset + localCol
globalRow = bandStartRow + localRow (your rowIndex is
maintaining that band).
```

3. Walking within a single 3×3 block

Let's step through the first block (top-left 3×3) to see your logic in action.

Assume at the start of block 1:

```
java
```

 Copy code

```
rowIndex = 0; // top band
colIndex = 0; // leftmost block
rowCount = 0; colCount = 0;
```

Now go through the first row of temp :

$k = 0$ (first cell in block row) $k == 0 \rightarrow$

- offset = colIndex = 0 .
- Place: `nineByNine[0][0] = temp[0][0]` .
- `colCount++ \rightarrow 1` `colIndex++ \rightarrow 1`
-

Now we're at $(\text{rowCount}, \text{colCount}) = (0,1)$ in temp and $(\text{rowIndex}, \text{colIndex}) = (0,1)$ in the big grid.

$k = 1$ $k \neq 0$, so leave offset as 0.

Place: `nineByNine[0][1] = temp[0][1]` .

- `colCount++ \rightarrow 2`
- `colIndex++ \rightarrow 2` k
-
- $= 2$
- Place: `nineByNine[0][2] = temp[0][2]` .
- `colCount++ \rightarrow 3`
- `colIndex++ \rightarrow 3`

Now we've filled the first row of the block: $(0,0),(0,1),(0,2)$ mapped to $(0,0),(0,1), (0,2)$.

At this point, because `colCount \geq 2` and `rowCount \neq 2` , your outer if kicks in:

```
java
```

 Copy code

```
if (colCount  $\geq$  2 && rowCount  $\neq$  2) {
    rowCount++; // 1    colCount = 0;
    rowIndex++; // 1

    if (offset == 0) {
        colIndex = 0; // reset to the block's starting column
    }
}
```

So:

- Local: rowCount moves to 1, colCount back to 0 → second row of 3×3 .
- Global: rowIndex moves to 1, colIndex resets to 0 → start of the second row in the same 3-wide block.

Next row of the mini-block will fill:

• $(1,0) \rightarrow (1,0)$, $(1,1) \rightarrow (1,1)$, $(1,2) \rightarrow (1,2)$, and similarly for the third row, because you repeat the same pattern with offset and row increment.

So within one 3×3 :

- rowCount / colCount ensure you visit all 9 cells in temp,
- rowIndex / colIndex + offset ensure those 9 cells land in a 3×3 rectangle in the 9×9 at the correct columns.

4. Jumping between 3×3 blocks in a band

After you've processed all 9 cells in a block, numberOf3x3Processed == 9 .

Then you check:

java

 Copy code

```
if (colIndex == 8 && numberOf3x3Processed == 9) {
    // End of band horizontally
    colIndex = 0;    rowIndex = rowIndex +
3;
}

if (colIndex != 8 && numberOf3x3Processed == 9) {
    // Still within the band    colIndex =
colIndex + 1;    rowIndex = 0;
}
```

The idea here:

- If the block you just finished ended at the right edge of the band (col 8):
 - You've placed block 3, 6, or 9 in that row band,
 - So the next block should be in the band below → rowIndex += 3 , colIndex = 0 .
- If you ended before col 8:
 -
 -
 -

You've placed block 1 or 2 in the current band, So
the next block should be to the right:

You bump colIndex by 1 (so next 3×3 will effectively start shifted right), And reset
rowIndex to 0 (top row of the band).

Together with offset logic, that gives you:

- Block 1 at columns 0–2,
- Block 2 at columns 3–5,
- Block 3 at columns 6–8,
- Then drop down 3 rows and repeat.

It's just that you implemented it incrementally, via movement, instead of with a closedform formula.

5. Conceptual summary of your fill logic

Your mental model for filling is:

1. Treat the big board as 9 slots for 3×3 blocks.
2. For each 3×3 block:
 - Use rowCount / colCount to walk the block locally (0..2, 0..2).
 - Use rowIndex / colIndex to walk the corresponding region in the big board. Use offset to
 - remember “where this block starts horizontally” inside its row band.
3. When you move to a new row of the 3×3:
 - Increase rowIndex globally,
 - Reset colIndex to offset so you're still inside the same block region.
4. When a 3×3 is finished:
 - If you're mid-band: jump right to the next block.
 - If you're at the band edge: jump down to the next band.

That's exactly what I meant by “simulating movement over the grid”:

you guide a cursor (rowIndex, colIndex) around, and your offset is the anchor that keeps each 3×3 aligned properly when you drop each row of that block.