📦 What is an Array?

An array is a collection of elements of the same type, stored in contiguous memory locations. You access each element using its index.

Example: `int[] arr = {10, 20, 30, 40};`

🧠 **Real-Life Analogy**: A row of **tiffin boxes** on a kitchen shelf—each box holds one dish, and you can pick any box by its position.

🔍 Indexing in Arrays

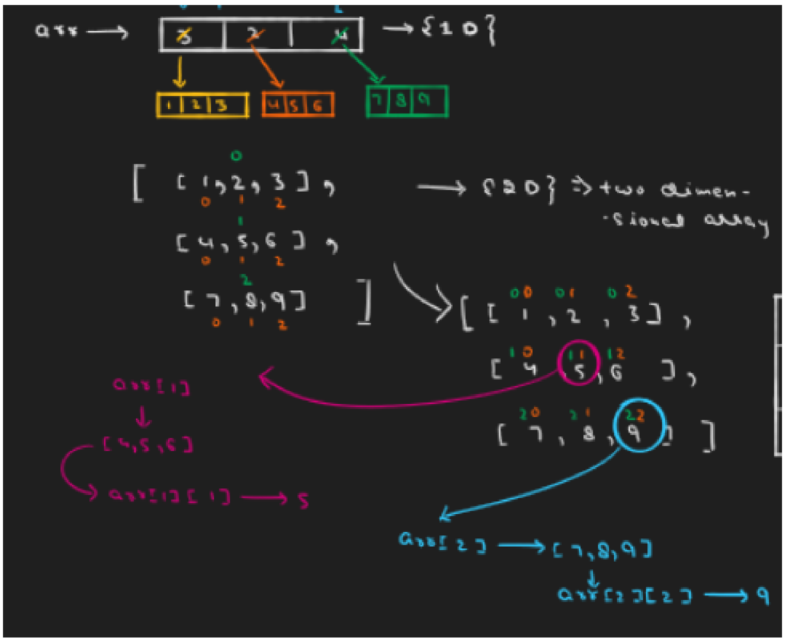
- Java indexing starts from **0**
- `arr[0]` → First element
- `arr[1]` → Second element

Common Mistake: Accessing `arr[4]` when array size is 4 throws an error (`ArrayIndexOutOfBoundsException`).

🧮 What is a Matrix?

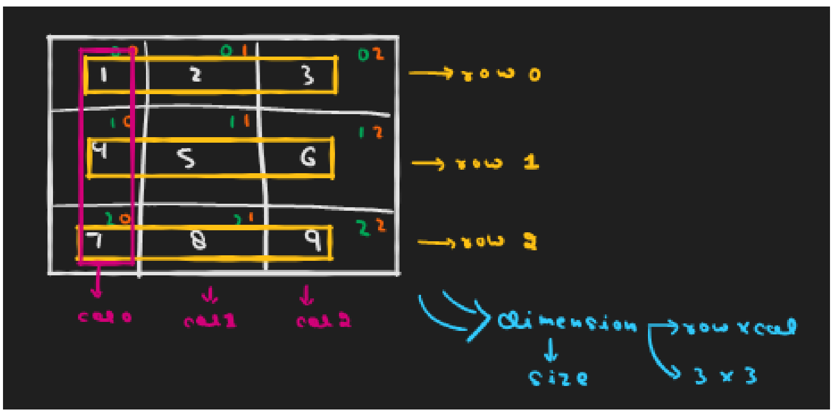
A 2D array in Java is a data structure that stores elements in a **grid-like format**, consisting of **rows and columns**. It is essentially an array of arrays, where each element of the outer array is itself an array.

A matrix is a 2D array—data arranged in rows and columns.



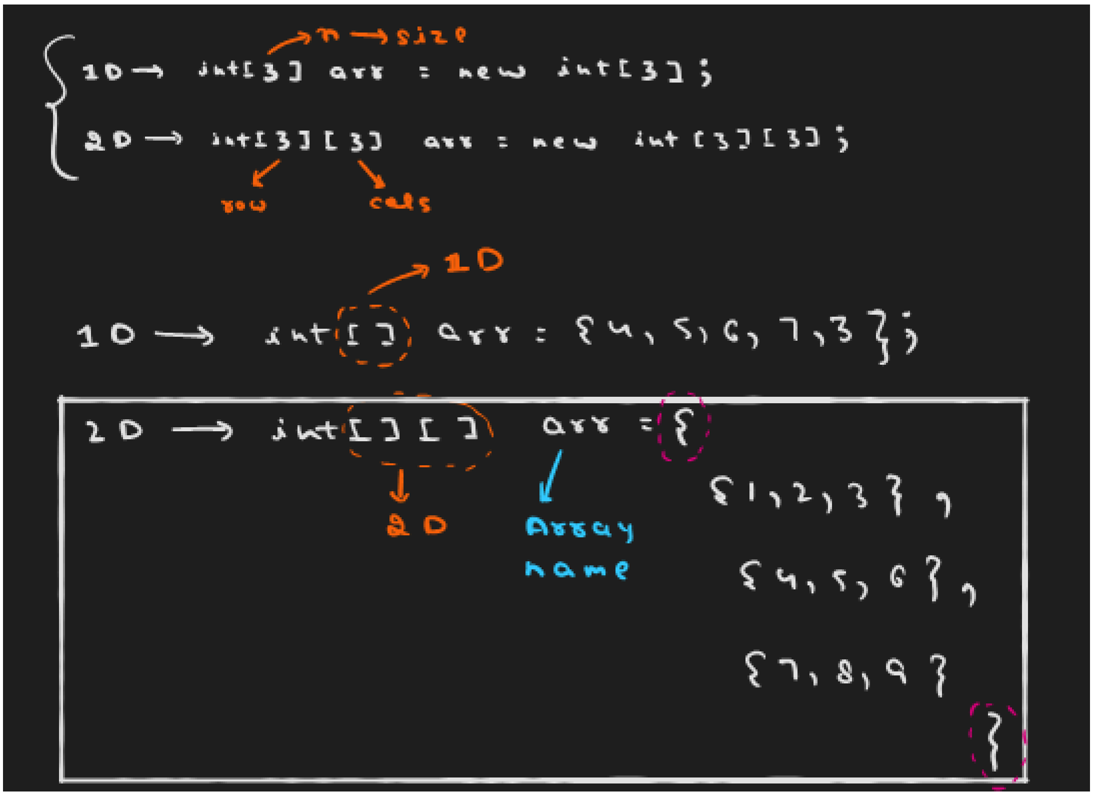
🧭 Rows vs Columns

- A **row** is a horizontal line of elements.
- A **column** is a vertical line of elements.



🔍 Indexing in 2D Arrays

- `matrix[0][2]` → Row 0, Column 2 → Value: 3
- `matrix[2][0]` → Row 2, Column 0 → Value: 7



Example:

```
int[][] matrix = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

🧠 **Real-Life Analogy**:

- Think of a classroom seating chart—rows are benches, columns are students per bench
- Marks of students in different subjects.
- Representing tabular data (e.g., marksheets, price charts)
- Mathematical matrices
- Grids in games or simulation

🧵 Memory Model in Java

- Arrays are stored in **heap memory**
- 2D arrays are arrays of arrays

✅ Pros of Arrays

- Fast access using index
- Simple to implement
- Memory-efficient for fixed-size data

⚠️ Cons of Arrays

- Fixed size (can't grow dynamically)
- Insertion/deletion is costly

- Only stores one type of data

🧠 When to Use Arrays

Use arrays when:

- You know the size in advance
- You need fast access by index
- You're working with homogeneous data

Avoid arrays when:

- You need dynamic resizing (use ArrayList)
- You want to store mixed types (use classes or maps)

Real-Life Analogy:

- Use ArrayList for **street food orders** that change every minute.
- Use arrays for **fixed menu items** in a thali.

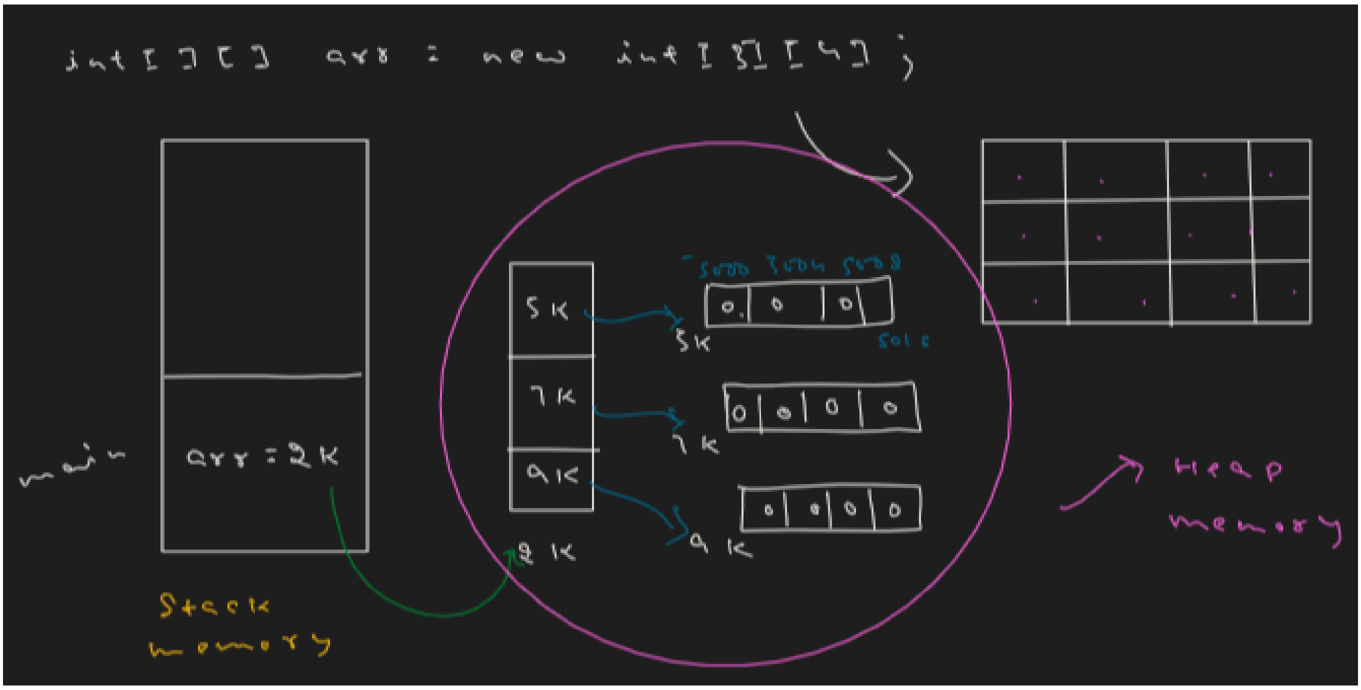
🧠 Clarifying Memory Allocation in 2D Arrays (Java)

❌ **Misconception:**

> "2D arrays in Java are stored in contiguous memory."

✅ **Reality:**

In Java, **2D arrays are not stored in contiguous memory**. Instead, they are implemented as **arrays of arrays**. Each row is a separate 1D array, and these row arrays may be stored at different memory locations.



📦 What Actually Happens:

int[][] arr= new int[3][34];

- `arr` is a reference to an array of 3 elements.
- Each `arr[i]` is a reference to a separate 1D array of 4 integers.
- These row arrays are individually allocated in heap memory.

🧵 Analogy:

Imagine a **train** with 3 coaches:

- The train (`arr`) knows it has 3 coaches.

- Each coach (`arr[i]`) is built separately and may be parked at different locations in the yard.
- The train links them together, but they're not physically connected in memory.

🔍 Contrast with C/C++:

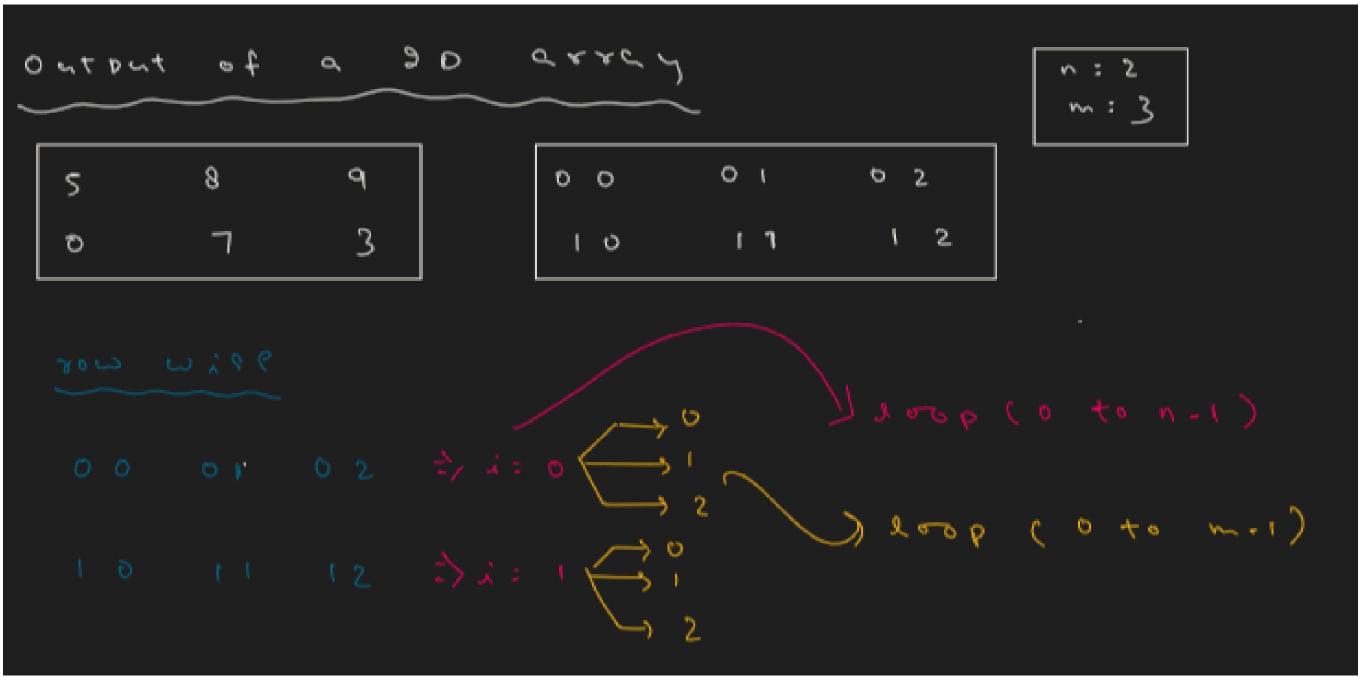
In C/C++, a 2D array like `int arr[3][4]` is stored in **contiguous memory**, laid out row by row. Java does **not** follow this model.

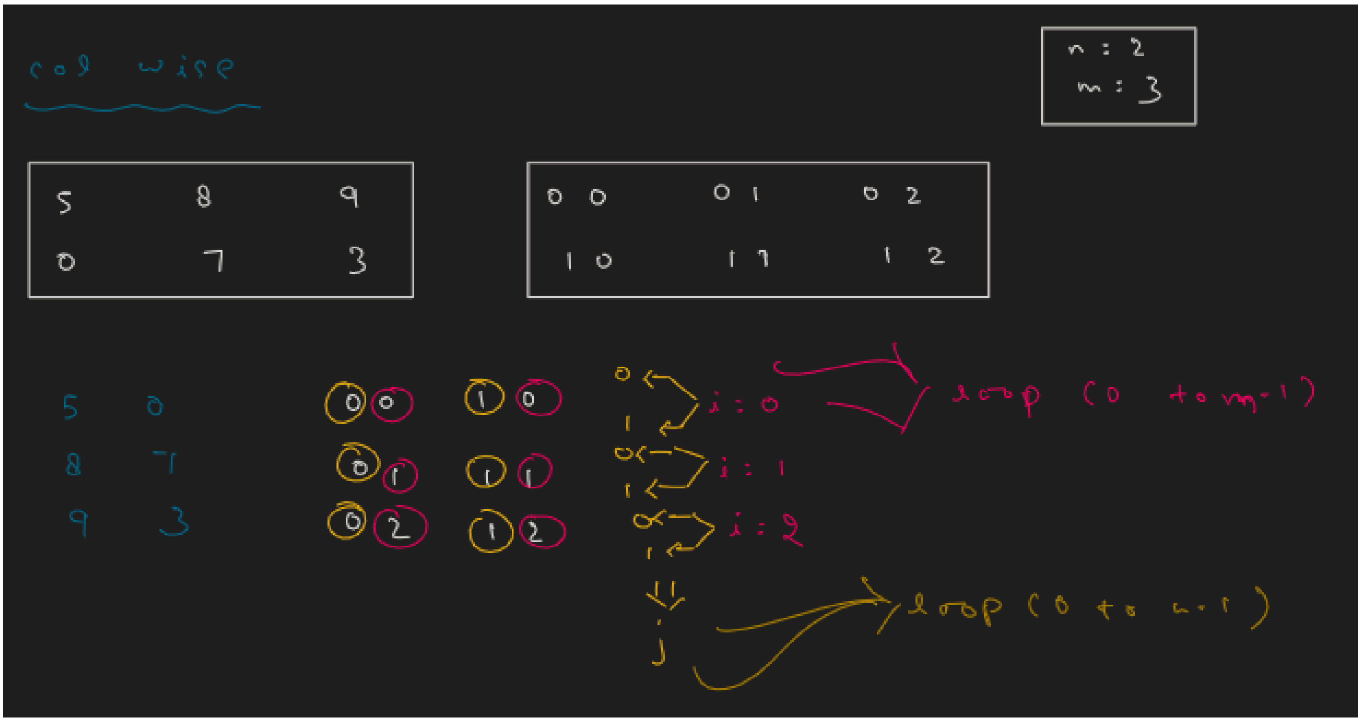
int [][] arr = new int [n][m];

n . 1 ⇒ Total no. of 1D array .

```java
public class Main {

      public static void main(String[] args) {
            int[][] arr = new int[3][4];
            System.out.println(arr); //[[I@12a3a380
            System.out.println(arr[1]); //[I@29453f44
            System.out.println(arr[1][2]); //0

            int[][] other = arr;
            // Will it form a 2D array ? No only address assignment will happen

            // row length
            int row = arr.length; // 3
            System.out.println(row);

            // col length
            int col = arr[0].length; // 4
            System.out.println(col);
      }

}
```

🎯 Input & Output for a 2D Array in Java





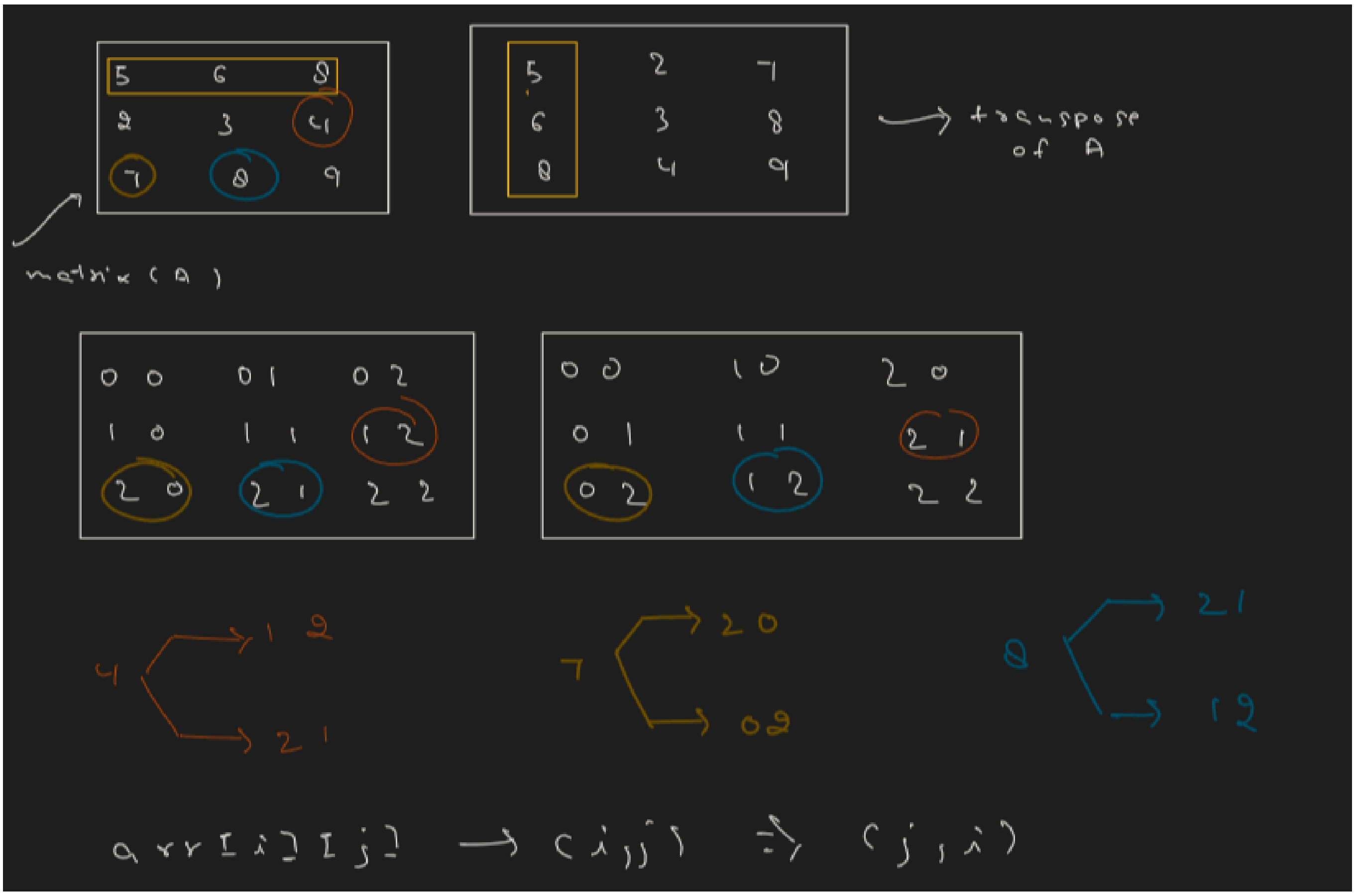

```java
public class Main {

    public static void main(String[] args) {

      Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(); // row
        int m = sc.nextInt(); // col
        int[][] arr = new int[n][m]; // [row][col]

        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[0].length; j++) {
                arr[i][j] = sc.nextInt();
            }
        }
```

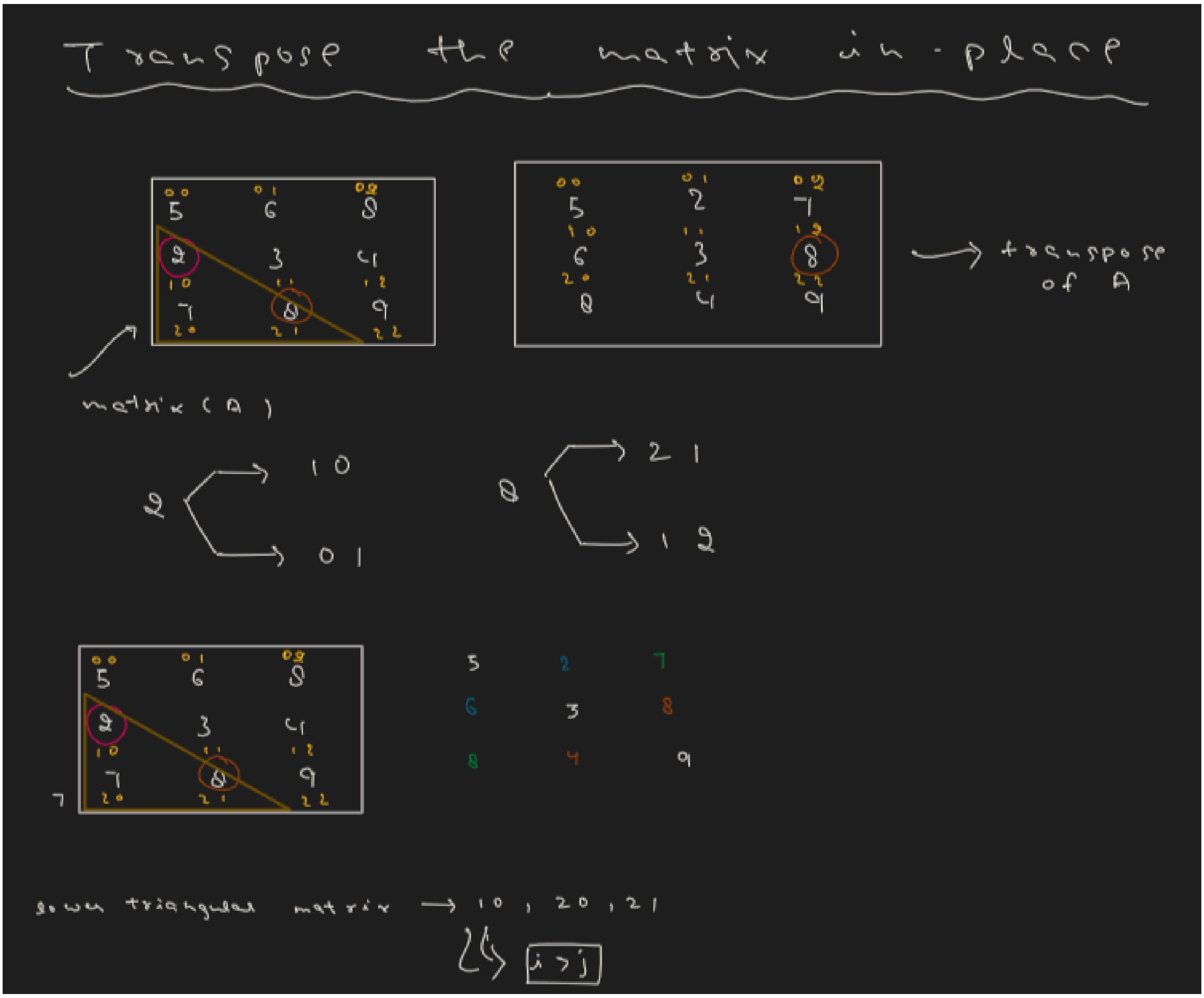
```java
        displayRowWise(arr);
        displayColWise(arr);
    }

    public static void displayRowWise(int[][] arr) {
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[0].length; j++) {
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();
        }
    }

    public static void displayColWise(int[][] arr) {
        for (int i = 0; i < arr[0].length; i++) {
            for (int j = 0; j < arr.length; j++) {
                System.out.print(arr[j][i]+" ");
            }
            System.out.println();
        }
    }

}
```

🎯 Wave Print -> Row & Col Wise

//To-Do

🎯 Transpose of a Square Matrix - O(n*m)





```java
public class Main {

    public static void main(String[] args) {
        Scanner scn=new Scanner(System.in);
        int n=scn.nextInt();

        int a[][]=new int[n][n];

        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                a[i][j]=scn.nextInt();
            }
        }

        int[][] trans = transpose(a);
        print2DArray(trans);

        transpose2(a);
        print2DArray(a);
    }

    static void swap(int a[][],int i,int j){
        int temp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = temp;
    }

    static void print2DArray(int[][] arr){
        int n = arr.length;

        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();
        }
    }

     static int[][] transpose(int[][] arr){
        int n = arr.length;

        int[][] trans = new int[n][m];

        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                trans[i][j] = arr[j][i];

        return trans;
    }

    static void transpose2(int[][] arr){
        int n = arr.length;

        int[][] trans = new int[n][n];
```

```java
        for(int i=0;i<n;i++)
            for(int j=0;j<=i;j++)
                swap(arr,i,j);
    }

}
```

🎯 Search a 2D Matrix II

https://leetcode.com/problems/search-a-2d-matrix-ii/

🔍 Linear Search Solution - O(n*m)

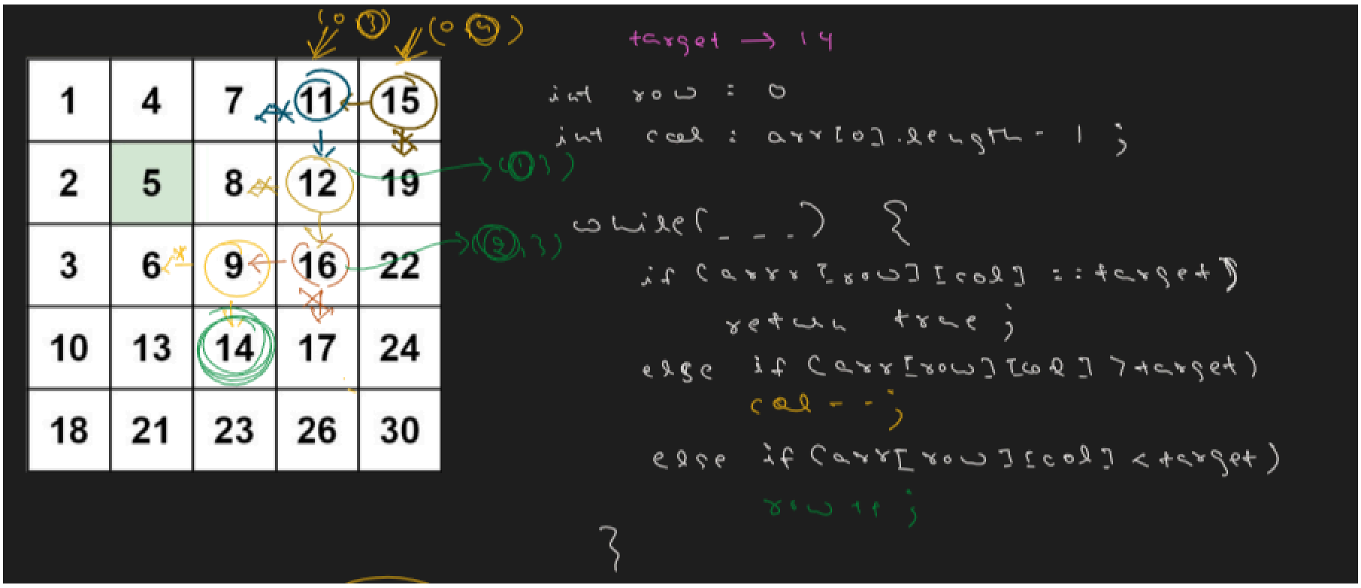
```java
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {

        int row = matrix.length;
        int col = matrix[0].length;

        for(int i=0; i<row; i++){
            for(int j=0; j<col; j++){
                if(matrix[i][j] == target){
                    return true;
                }
            }
        }
        return false;

    }
}
```

🧠 Binary Search Solution

❌ Apply binary search on all 1D arrays → **O(n log m) Solution**

What I can see very clearly is that if I start from the **top-right element**, **every element below it is greater, and every element to the left is smaller.**

So, if I am searching for an element like 14 in the given example, and the current element is larger than 14, I will move left. This effectively reduces the size of the matrix I need to search. I will continue reducing the search space in this way until I either find the target element or reach the end of the matrix.



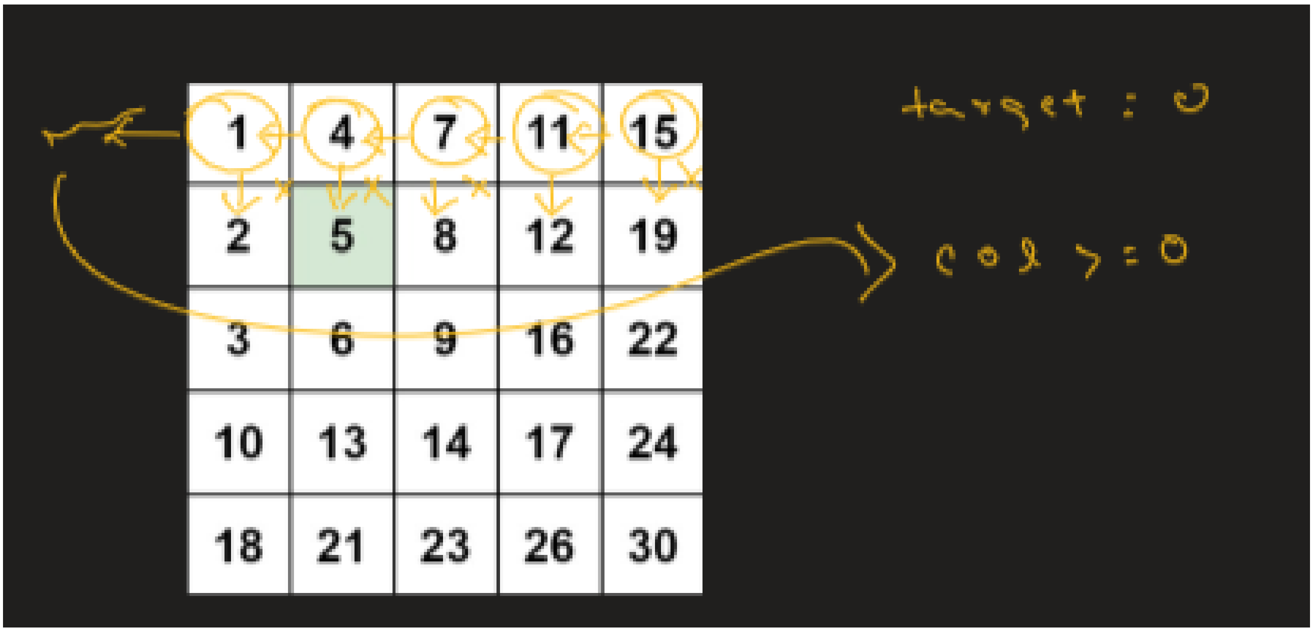
To determine the **loop's exit conditions**, let's consider an example: suppose I am **searching for the value 20**, which I can clearly see is **not present** in the matrix.

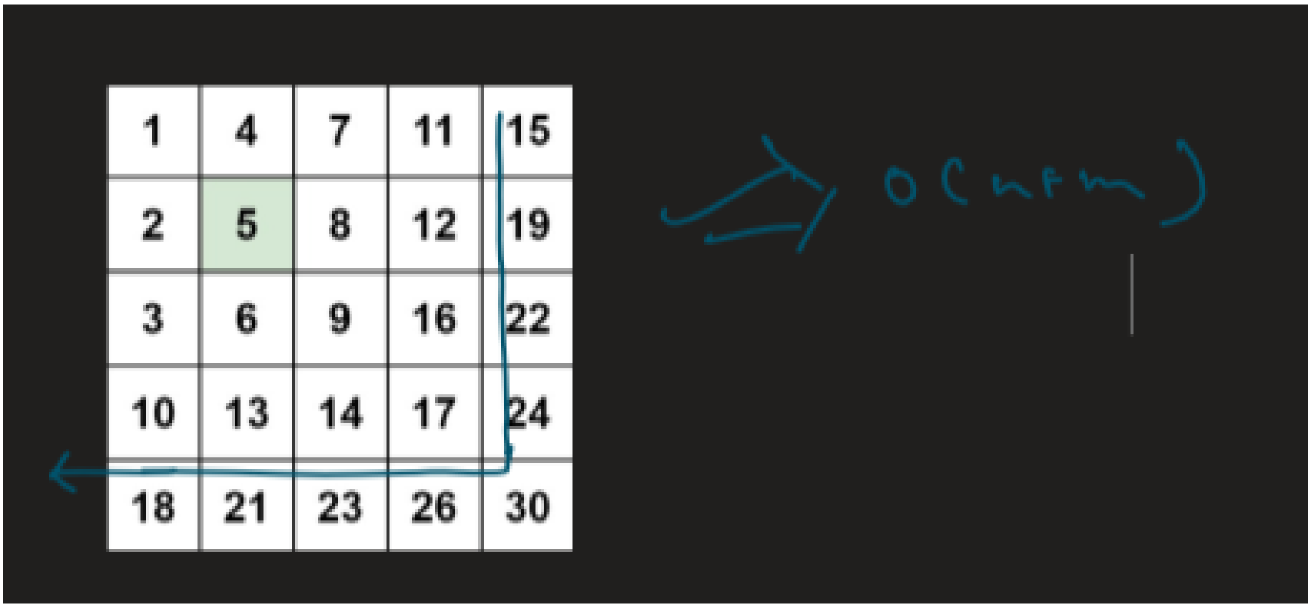
I would start from the top-right element (e.g., 15), and since 20 is greater than 15, I would move down. I continue this process—moving down if the target is greater, or left if it's smaller—until I either find the target or **move out of the matrix bounds**.

There are two ways I can go out of the matrix:

1. **Downward** — when my `row` index becomes equal to the number of rows in the matrix (i.e., `row == matrix.length`).
2. **To the left** — when my `col` index becomes less than 0 (i.e., `col < 0`).

Either of these conditions means I've exhausted all possibilities, and the element is not in the matrix.






```java
public class Main {

    public static void main(String[] args) {
        int[][] arr = {
        { 1, 4, 7, 11, 15 },
        { 2, 5, 8, 12, 19 },
        { 3, 6, 9, 16, 22 },
        { 10, 13, 14, 17, 24 },
        { 18, 21, 23, 26, 30 }
     };
        int item = 23;
        System.out.println(search(arr, item));
    }

    public static boolean search(int[][] arr, int item) {
        int row=0;
        int col = arr[0].length-1;

        while(row<arr.length && col>=0) {
            if(arr[row][col]==item) {
```

```
                        return true;
                    }
                    else if(arr[row][col]>item) {
                        col--;
                    }
                    else {
                        row++;
                    }
                }
                return false;
        }

}
```

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {

        int row = 0;
            int col = matrix[0].length-1;

            while(row < matrix.length && col >= 0) {
                if(matrix[row][col] == target) {
                    return true;
                }
                else if(matrix[row][col] > target) {
                    col--;
                }
                else {
                    row++;
                }
            }
            return false;

    }
}
```

🔍 Why Starting from the Top-Right Corner Works Best

Now, let's discuss why starting from the **top-right corner** is the best choice among the four matrix corners.

Let's begin with the **top-left corner**. If you look at this position, moving either **down** or **right** will always lead you to elements **greater** than the current element. Therefore, you can't confidently eliminate any row or column based on comparison, making it unsuitable for our logic.

The same applies to the **bottom-right corner**. If you move **up** or **left**, you'll encounter elements that are **smaller**, so again, it's not helpful for narrowing the search space in a deterministic way.

Now let's talk about the **bottom-left corner** — the only other one we haven't explored. From this position:

- If the target is **greater** than the current element, you can move **right**.
- If the target is **smaller**, you can move **up**.

This logic is also valid, and just like the top-right corner, it allows us to systematically eliminate rows or columns. So technically, we could also start from the bottom-left.

With that understanding, we can now write the solution in Java.

```java
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {

        int row = matrix.length - 1;  // Start from the bottom-left corner
        int col = 0;

        while (row >= 0 && col < matrix[0].length) {
            int current = matrix[row][col];

            if (current == target) {
                return true;
            } else if (current > target) {
                row--; // Move up
            } else {
                col++; // Move right
            }
        }

        return false;
    }
}
```