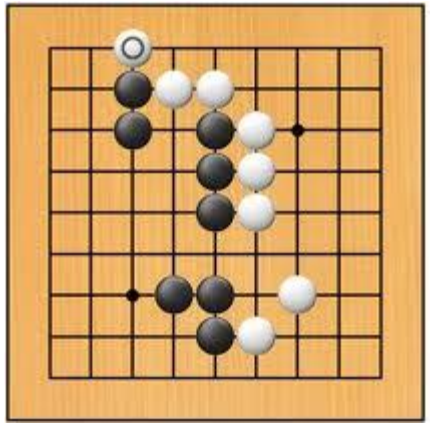
COMP 1020 - Multidimensional Arrays

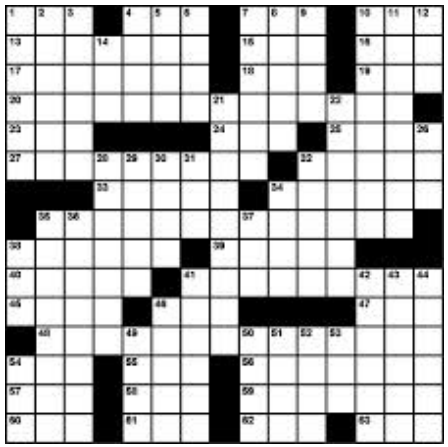
UNIT 7

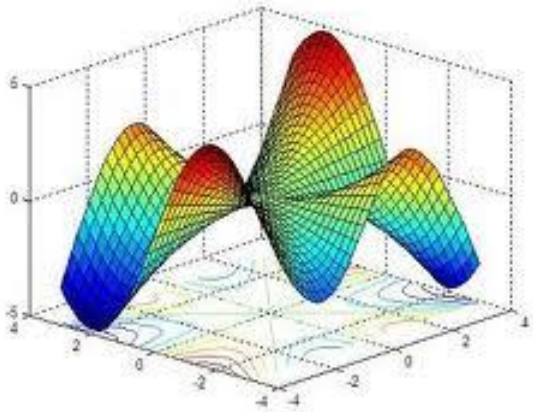
Multidimensional arrays

- Often, a matrix (2D) of data is needed, organized into rows and columns:

$$A = \begin{bmatrix} 7 & 2 & -1 \\ 4 & 1 & 0 \\ 5 & 3 & 6 \end{bmatrix}$$







Multidimensional arrays

- Sometimes, you might need more dimensions than 2
- Let's say you want to record the temperature in Winnipeg:
  - For each month in a year
  - For each day of each month
  - For each hour of each day

3 dimensions:
month x day x hour

Multidimensional arrays

- Sometimes, you might need more dimensions than 2
- Let's say you want to record the temperature in Winnipeg:
  - For each month in a year
  - For each day of each month
  - For each hour of each day

  3 dimensions:
  month x day x hour

  - Let's say now you're interested in those measurements at 5 different locations in the city: that's a fourth dimension

Arrays of arrays

- In Java, you can have an array of any type of data, including another array:

```
any-type[ ] x = new any-type[n]; //n any-type's
```

  - "int[ ]" is a type, so you could have:

```
int[ ][ ] x = new int[3][3]; //a 3x3 array
```

Arrays of arrays

- In Java, you can have an array of any type of data, including another array:

```
any-type[ ] x = new any-type[n]; //n any-type's
```

  - "int[ ]" is a type, so you could have:

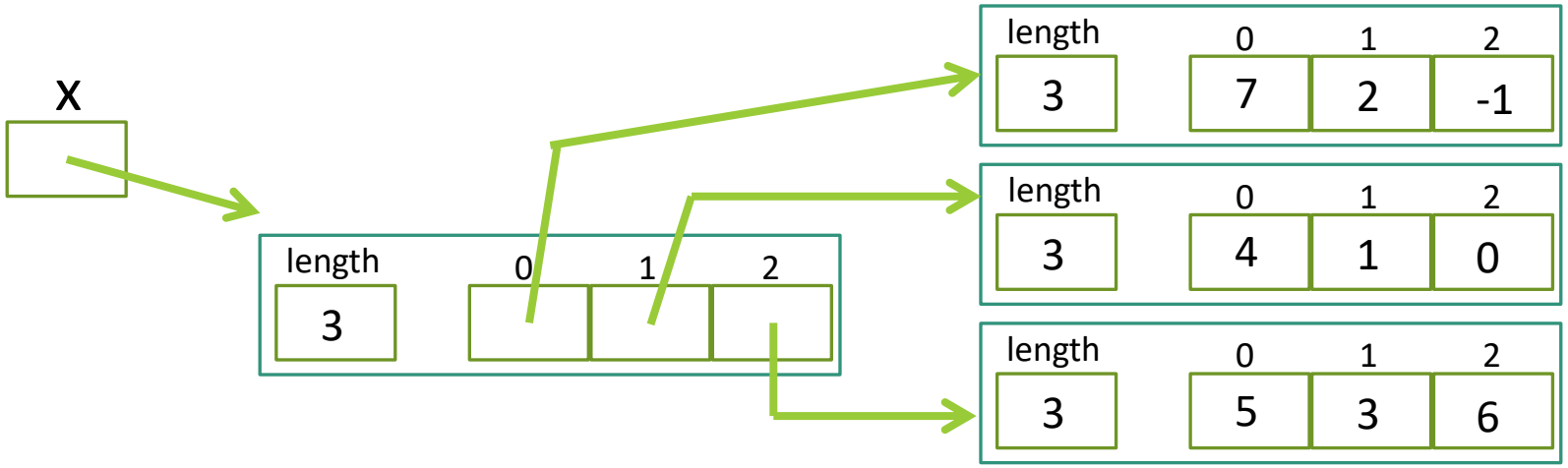
```
int[ ][ ] x = new int[3][3]; //a 3x3 array
```

Just like int[] meant an array of ints, int[][] means an array of arrays of ints

Arrays of arrays

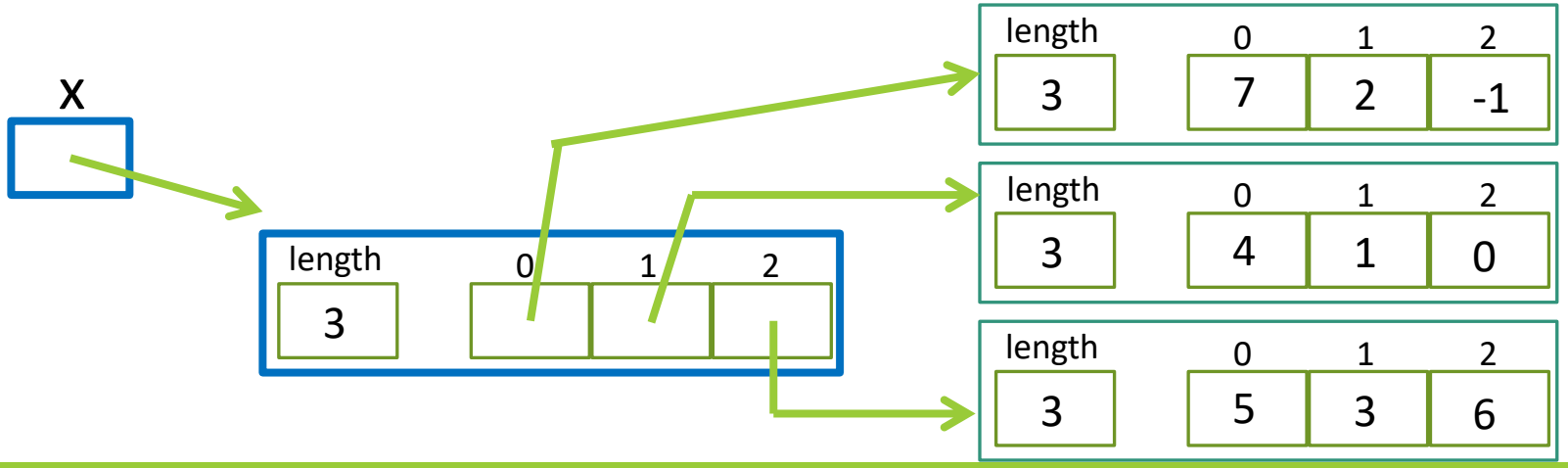
```
int[ ][ ] x = new int[3][3]; //a 3x3 array
```

- So, this is stored as a (reference to) an array of 3 things, each of which is a (reference to) an array of 3 int values



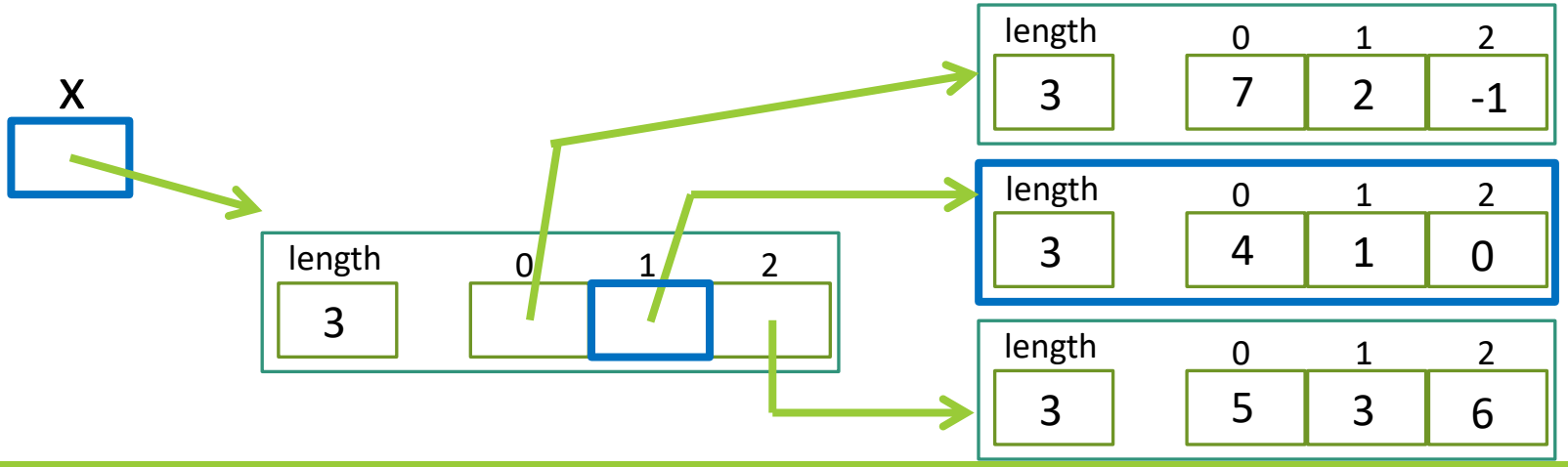
Accessing elements or rows

- You use subscripts as usual to access particular elements of these arrays:
  - x gives you the whole array



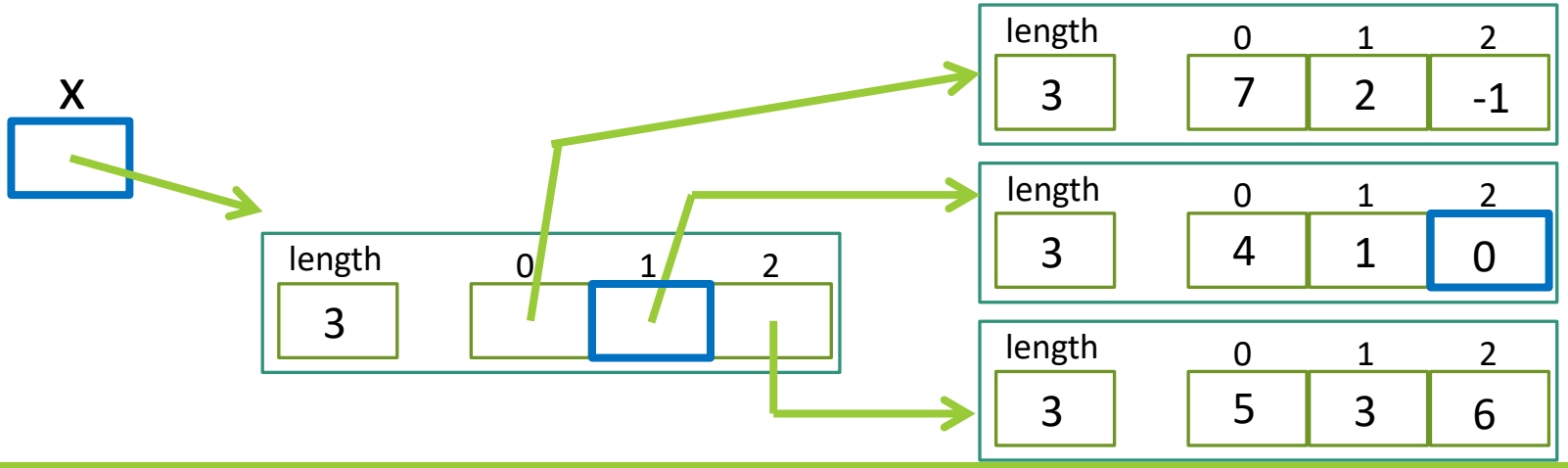
Accessing elements or rows

- You use subscripts as usual to access particular elements of these arrays:
  - x gives you the whole array
  - x[1] gives you one particular "row"



Accessing elements or rows

- You use subscripts as usual to access particular elements of these arrays:
  - x gives you the whole array
  - x[1] gives you one particular “row”
  - x[1][2] gives you one particular element in “row” 1
    - Same as (x[1])[2]



Another way to see it

- Usually, for an array of arrays of ints, you can just imagine a 2D matrix

- Example:

```
int[ ][ ] a = new int[3][3];
a[0][0] = 5;
a[0][2] = -3;
a[1][2] = 7;
a[2][0] = 9;
```

Another way to see it

- Usually, for an array of arrays of ints, you can just imagine a 2D matrix

- Example:

```
int[ ][ ] a = new int[3][3];
a[0][0] = 5;
a[0][2] = -3;
a[1][2] = 7;
a[2][0] = 9;
```

	0	1	2
0	5	0	-3
1	0	0	7
2	9	0	0

Another way to see it

- Usually, for an array of arrays of ints, you can just imagine a 2D matrix

- Example:

```
int[ ][ ] a = new int[3][3];
a[0][0] = 5;
a[0][2] = -3;
a[1][2] = 7;
a[2][0] = 9;
```

When thinking of it as a matrix, a[r][c] is the element in row r and column c.

	0	1	2
0	5	0	-3
1	0	0	7
2	9	0	0

A warning

- A 2D array is not actually represented graphically in the memory → it is stored as an array of arrays, just like we've seen in the previous slides

A warning

- A 2D array is not actually represented graphically in the memory → it is stored as an array of arrays, just like we've seen in the previous slides

- Most people tend to mentally represent the 2D matrix in this way:

number of rows

```
int[ ][ ] a = new int[2][3];
```

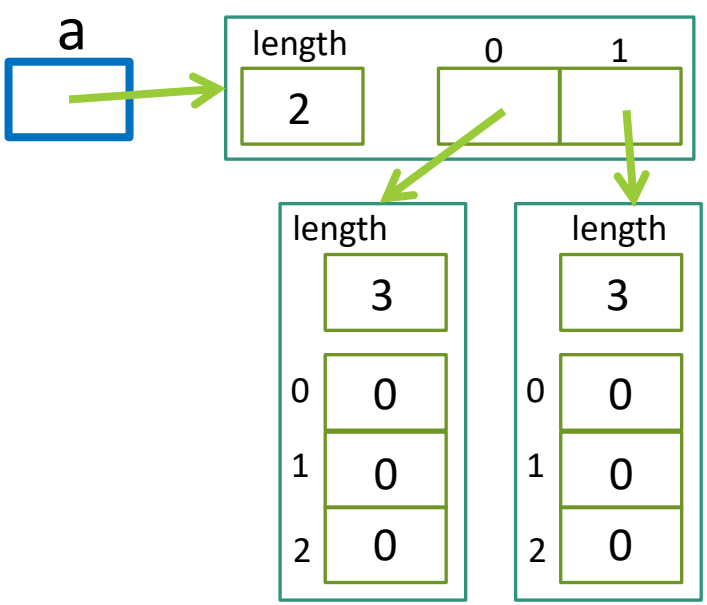
number of columns

	0	1	2
0	0	0	0
1	0	0	0

A warning

- But technically there are no “rows” and “columns”, it’s just a mental representation → you could look at it both ways

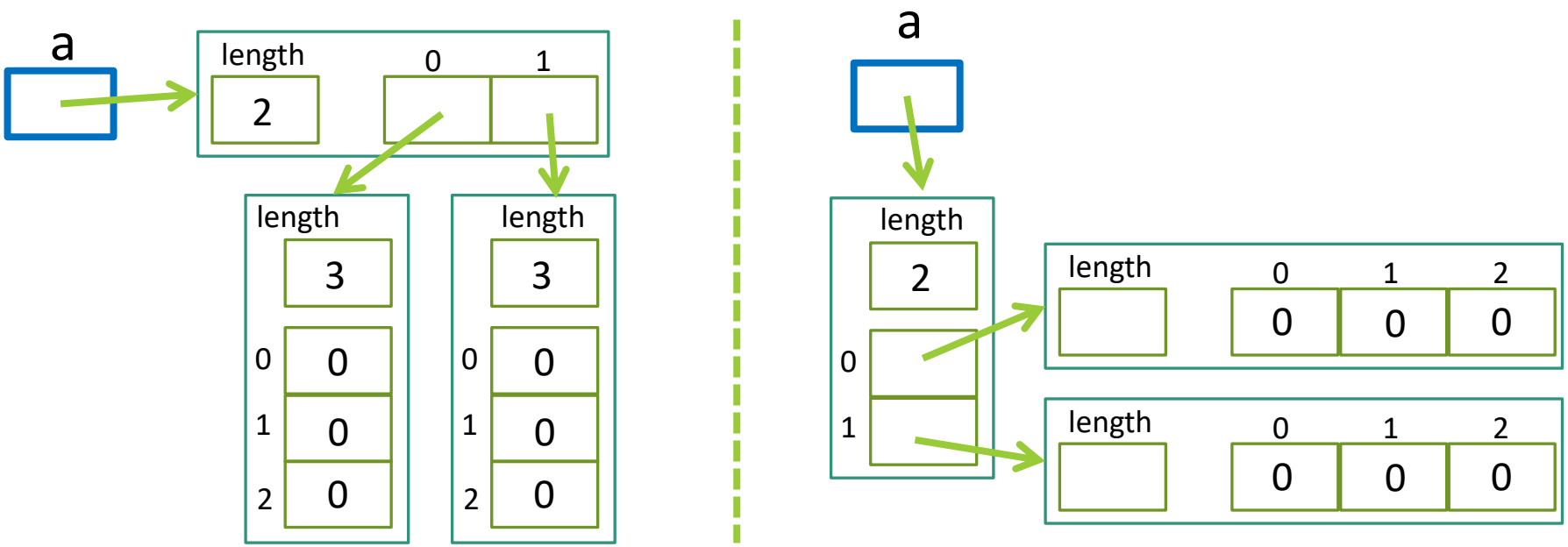
int[ ][ ] a = new int[2][3];



A warning

- But technically there are no “rows” and “columns”, it’s just a mental representation → you could look at it both ways

int[ ][ ] a = new int[2][3];



A warning

- When dealing with more than 2 dimensions, you can't represent the dimensions as rows and columns anyway
- Just make sure you remember which dimension is which when you declare and use your multidimensional array…
- … and always remember that a multidimensional array is an array of arrays of [anything]

Creating and accessing an array

- Creating a 2D array of Strings, 4 by 10:

```
String[][] myArray = new String[4][10];
```

- Access order is consistent with creation order (for the dimensions):
  - Printing “row” 0 and “column” 9 (remember we start counting from 0)

```
System.out.println(myArray[0][9]);
```

Initializing with { }

- You can use the { } to initialize a multidimensional array:

```
int[ ][ ] a = { {1,2,3}, {4,5,6}, {7,8,9} };
int[ ][ ][ ] a2 = { {{1,2},{3,4}} , {{5,6},{7,8}} , {{9,8},{7,6}} };
```

Initializing with { }

- You can use the { } to initialize a multidimensional array:

```
int[ ][ ] a = { {1,2,3}, {4,5,6}, {7,8,9} };
int[ ][ ][ ] a2 = { {{1,2},{3,4}} , {{5,6},{7,8}} , {{9,8},{7,6}} };
```

- The “rows” don’t even need to be the same size:

```
int[ ][ ] a = { {1,2} , {1,2,3,4,5,6}, {1} };
```

Using new

- When you use new, like this:

```
int[ ][ ] a = new int[2][4];  //2 arrays, each an int[4] object
```

- You can leave the smaller arrays unallocated (the last dimensions only, i.e. you must give the first $k \geq 1$ sizes, then you can leave the rest blank):

```
int[ ][ ] a = new int[2][ ]; //2 unspecified arrays
                             //The 2 entries will be null
int[][][][][] a = new int[3][5][][][];  //this works too
```

Let’s practice using 2D arrays

- We will now do some coding

Let’s practice using 2D arrays

- We will now do some coding
- Let’s build a CharMatrix object, which will contain a 2D matrix of chars as the instance variable

Let’s practice using 2D arrays

- We will now do some coding
- Let’s build a CharMatrix object, which will contain a 2D matrix of chars as the instance variable
- We will add some constructors
  - initializing the size (nb of “rows” and “columns”)
  - filling up the matrix with a specific character → should be calling a separate method to do this

Let’s practice using 2D arrays

- We will now do some coding
- Let’s build a CharMatrix object, which will contain a 2D matrix of chars as the instance variable
- We will add some constructors
  - initializing the size (nb of “rows” and “columns”)
  - filling up the matrix with a specific character → should be calling a separate method to do this
- We will put a toString method (toStringRotated as well, to swap rows and cols, just for fun)

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

1) fillTopHalf(char)

x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

2) fillDiagonal(char)

x					
	x				
		x			
			x		
				x	
					x

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

3) fillAboveDiagonal(char)

x	x	x	x	x	x
	x	x	x	x	x
		x	x	x	x
			x	x	x
				x	x
					x

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

4) fillBelowDiagonal(char)

x					
x	x				
x	x	x			
x	x	x	x		
x	x	x	x	x	
x	x	x	x	x	x

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

5) fillEveryOtherRow(char)

x	x	x	x	x	x
x	x	x	x	x	x
x	x	x	x	x	x

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

6) fillEveryOtherCol (char)

x		x		x	
x		x		x	
x		x		x	
x		x		x	
x		x		x	
x		x		x	

Let’s practice using 2D arrays

- Then we’ll build a series of methods to fill the matrix with a certain character in different ways

7) fillChessBoard (char)

x		x		x	
	x		x		x
x		x		x	
	x		x		x
x		x		x	
	x		x		x

Let’s practice using 2D arrays

- This code is posted Moodle:
  - CharMatrix.java
  - CharMatrixTest.java

Creating and accessing an array

- Creating a 3D array of booleans, 2 by 3 by 4:

```
boolean[][][] myArray = new boolean[2][3][4];
```

- Access order is consistent with creation order (for the dimensions):
  - Changing a value to true at indices (1, 2, 3):

```
myArray[1][2][3] = true;
```

Creating and accessing an array

- Creating a 3D array of booleans, 2 by 3 by 4:

```
boolean[][][] myArray = new boolean[2][3][4];
```

- Access order is consistent with creation order (for the dimensions):
  - Changing a value to true at indices (1, 2, 3):

```
myArray[1][2][3] = true;
```

There are practically no limits to the number of dimensions for arrays: 2D, 3D, 4D, 5D, 6D, 7D, etc.

Creating and accessing an array

- Creating a 3D array of booleans, 2 by 3 by 4:

```
boolean[][][] myArray = new boolean[2][3][4];
```

- Access order is consistent with creation order (for the dimensions):
  - Changing a value to true at indices (1, 2, 3):

```
myArray[1][2][3] = true;
```

More precisely… there is a limit, but it's 255… you shouldn't have to worry about going over this limit…

Initializing with { }

- You can use the { } to initialize a multidimensional array:

```
int[ ][ ] a = { {1,2,3}, {4,5,6}, {7,8,9} };
int[ ][ ][ ] a2 = { {{1,2},{3,4}} , {{5,6},{7,8}} , {{9,8},{7,6}} };
```

Initializing with { }

- You can use the { } to initialize a multidimensional array:

```
int[ ][ ] a = { {1,2,3}, {4,5,6}, {7,8,9} };
int[ ][ ][ ] a2 = { {{1,2},{3,4}} , {{5,6},{7,8}} , {{9,8},{7,6}} };
```

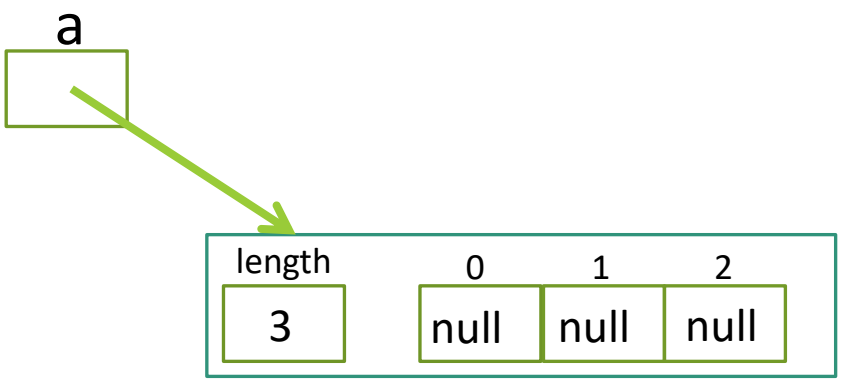
- The “rows” don’t even need to be the same size:

```
int[ ][ ] a = { {1,2} , {1,2,3,4,5,6}, {1} };
```

“Ragged” arrays

- You can create arrays containing smaller arrays of assorted sizes:

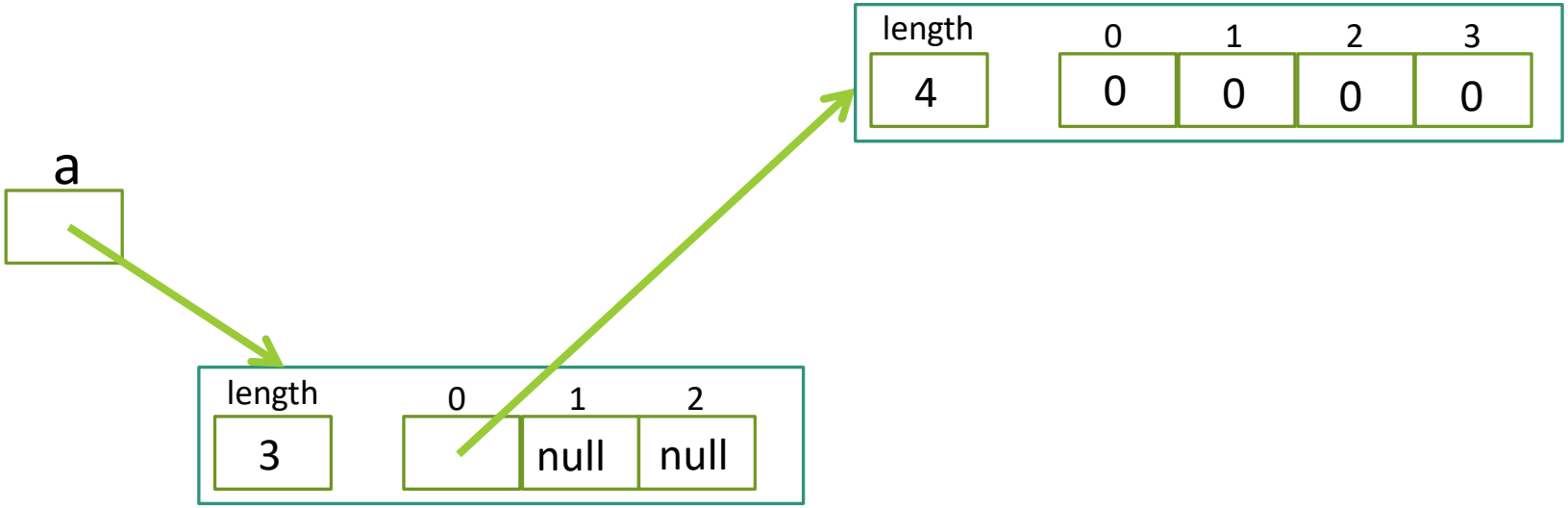
```
int[ ][ ] a = new int[3][ ];
```



“Ragged” arrays

- You can create arrays containing smaller arrays of assorted sizes:

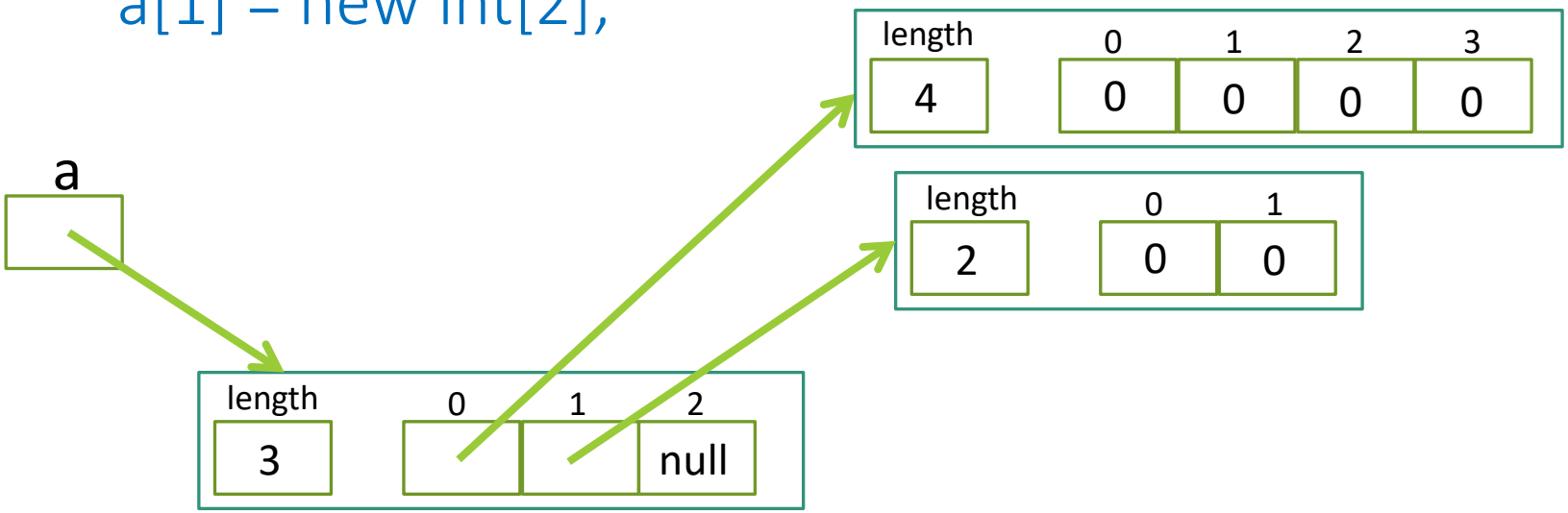
```
int[ ][ ] a = new int[3][ ];
a[0] = new int[4];
```



“Ragged” arrays

- You can create arrays containing smaller arrays of assorted sizes:

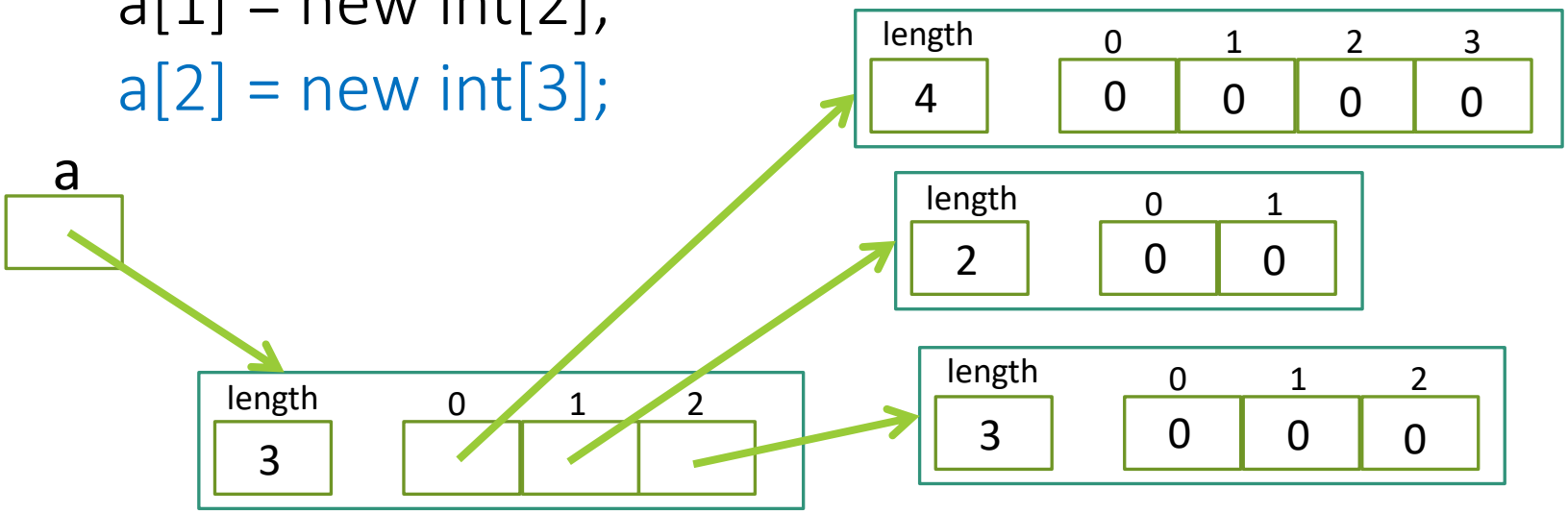
```
int[ ][ ] a = new int[3][ ];
a[0] = new int[4];
a[1] = new int[2];
```



“Ragged” arrays

- You can create arrays containing smaller arrays of assorted sizes:

```
int[ ][ ] a = new int[3][ ];
a[0] = new int[4];
a[1] = new int[2];
a[2] = new int[3];
```



Let's practice using 3D arrays

- Now let's build a different example, using 3D arrays
  - the temperature in Winnipeg example from the beginning of the slides

- The goal is to have an object, that I will call Weather, in which we will have the 3D array temperatures:
  - 1st dimension: month
  - 2nd dimension: day
  - 3rd dimension: hour

- Temperatures will be stored in ints

Let’s practice using 3D arrays

- Let’s use some static final constants to define some basic stuff:
  - an array for the number of days in each month
  - an array containing the names of the months
  - the number of hours in a day (24)
  - the hour that represents noon (12)

Let’s practice using 3D arrays

- Let’s build constructors
  - a constructor with no parameters (initializing the temperatures 3D array)
  - a constructor taking an array of average temperatures for each month (it will serve as a basis for generating random temperatures for the month), and then proceeds to generate random temperatures for each hour of each day in each month

Let’s practice using 3D arrays

- Let’s build static methods
  - getRandomIntAroundMean(int mean, int plusOrMinus): this method generates a random int, centered around “mean”, plus or minus a certain value
  - generateTempsForEachHour(int[] hourlyTemps, int meanDayTemp): this method fills up the hourlyTemps array with temperatures close to the meanDayTemp

Let’s practice using 3D arrays

- Let’s add some instance methods!
  - toString
  - getAvgTempDay(int month, int day): returns the avg temperature for the day (double)
  - getAvgTempMonth(int month): returns the avg temperature for the month (double)

Let’s practice using 3D arrays

- This code will be posted online:
  - Weather.java
  - WeatherTest.java