CS125 : Introduction to Computer Science

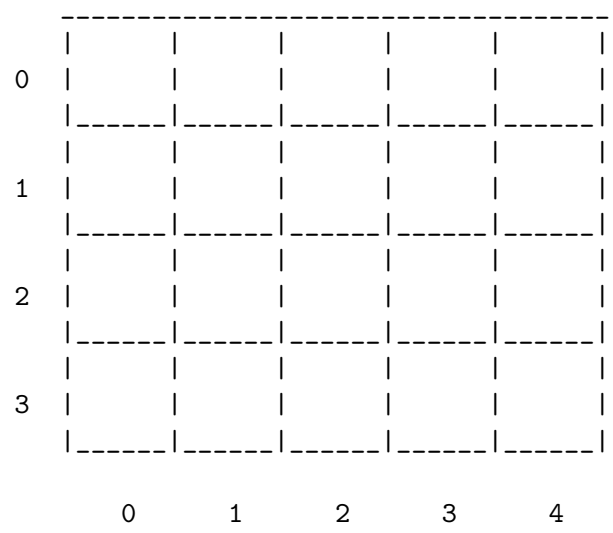
Lecture Notes #9
Multi-Dimensional Arrays

Lecture 9 : Multi-Dimensional Arrays

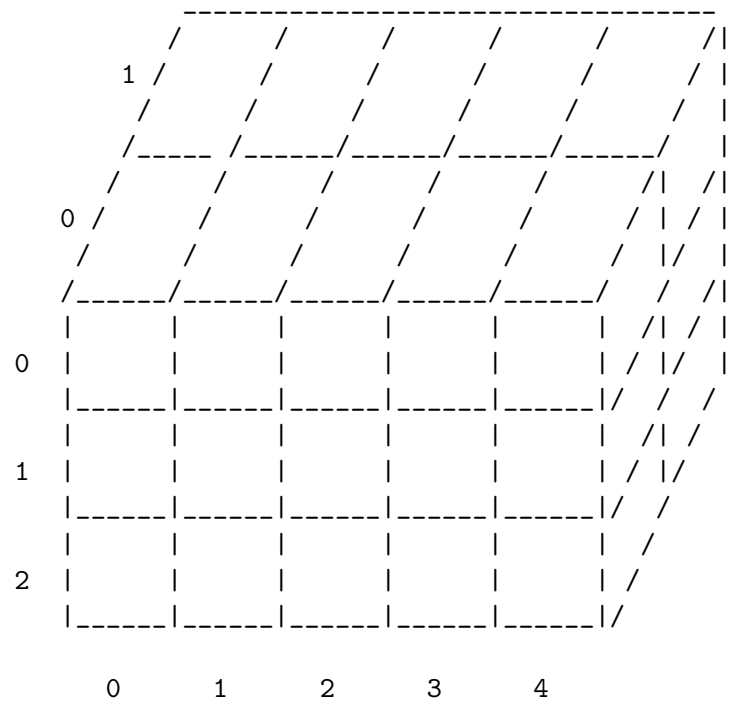
Multi-dimensional arrays

Up to now we have been talking about *one-dimensional arrays* (or alternatively, *single-dimensional arrays*. These arrays are by far the most common type of array, and so often we just say "array" when we mean this kind of array.

However, it is possible to have arrays of multiple dimensions as well. For example, if you would find it useful to have an array with both rows and columns (i.e. with two dimensions):



or an array with three dimensions:



then you can create such an array in Java!

The array creation syntax we discussed in the last notes packet – for one-dimensional arrays – was a special case of the more general array creation syntax. In general, the syntax for creating an array of `D` dimensions is as follows:

```
Type[][][]...[] varname = new Type[size1][size2][size3]...[sizeD];
    -----------                   -------------------------------
    D of these                        D of these
```

The `size1`, `size2`, etc. expressions in the square brackets above, are the sizes of the different dimensions in your array (for our purposes, those expressions will evaluate to positive integers; we will only allow the size to be `0` when dealing with one-dimensional arrays). For each dimension, the indices are a consecutive range of integers beginning at `0` and ending at `size - 1`, where `size` is the size of that dimension. So, for the array created by the above syntax, the first dimension has indices `0` through `size1 - 1`, the second dimension has indices `0` through `size2 - 1`, the third dimension would have indices `0` through `size3 - 1`, all the way up to dimension `D`, which has indices from `0` through `sizeD - 1`.

For multi-dimensional arrays, such as the two pictures you saw earlier, you decide which will be the first dimension, and which will the second dimension, and so on (if there are more than two dimensions), and then use the general syntax above. For example, for a two-dimensional array, we simplify the general syntax to two sets of brackets:

```
Type[][] varname = new Type[sizeOfFirstDimension][sizeOfSecondDimension];
```

In the picture of the two-dimensional array that we had earlier in this notes packet, if we wanted the number of rows in our picture to be the first dimension in our array (note there are four rows) and we wanted the number of columns in our picture to be the second dimension in our array (note there are five columns), and we wanted to hold values of type `int`, then we would write a statement such as the following one, in order to create that array:

```
int[][] theMatrix = new int[4][5];
```

and we have now created a two-dimensional array, with a variable name `theMatrix`, which has space for twenty integers. If we wanted to create an array to match the three-dimensional example from above, and we wanted the dimension with three cells (indexed with `0`, `1`, and `2`) to be the first dimension, and we wanted the dimension with five cells (indexed with `0`, `1`, `2`, `3`, and `4`) to be the second dimension, and we wanted the dimension with two cells (indexed with `0` and `1`) to be the third dimension, and wanted the array to hold values of type `char`, then in order to create such an array, we would use the following statement:

```
char[][][] threeDarray = new char[3][5][2];
```

Note that the syntax for one-dimensional arrays – which you learned in the last notes packet – is just the above syntax, with `D` being equal to `1`, since there is only one dimension. So, you would have exactly one pair of square brackets before the variable name, and one pair of square brackets with a size in it after the assignment operator – exactly as you saw in the previous notes packet:

```
Type[] varname = new Type[size1];
    --                   -------
  1 of these           1 of these
```

Accessing cells of multi-dimensional arrays

Just as the syntax for creating a one-dimensional array was a special case of the more general syntax for creating a `D`-dimensional array, the syntax for *accessing* a one-dimensional array is a special case of the more general syntax for *accessing* a `D`-dimensional array. Accessing a D-dimensional array is done with the following expression:

```
varname[index1][index2]...[indexD]
          --------------------------
               D of these
```

In the case of a two-dimensional array, you would have two sets of brackets:

```
varname[index1][index2]
```

In the case of a three-dimensional array, you would have three sets of brackets:

```
varname[index1][index2][index3]
```

and so on.
In the case of a one-dimensional array, that just simplifies to what you learned earlier, since there would be only one set of brackets:

```
varname[index1]
         --------
        1 of these
```

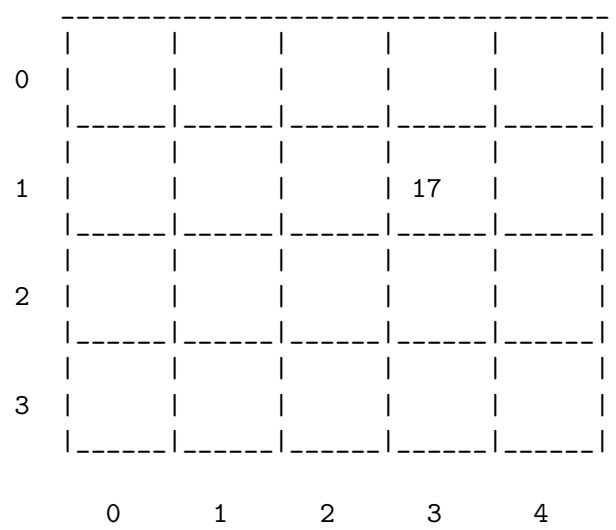
To give an example, if we want to access row `1`, column `3` of our two-dimensional array from the earlier examples (counting row `0` as the first row and column `0` as the first column, since all index ranges always start with `0`), we would use the following expression:

```
theMatrix[1][3]
```

So, we could write the value `17` into that cell using the following assignment statement:

```
theMatrix[1][3] = 17;
```

and that would give us the following picture:



If we then wanted to print out the value of that cell to the screen, we would use the following statement"

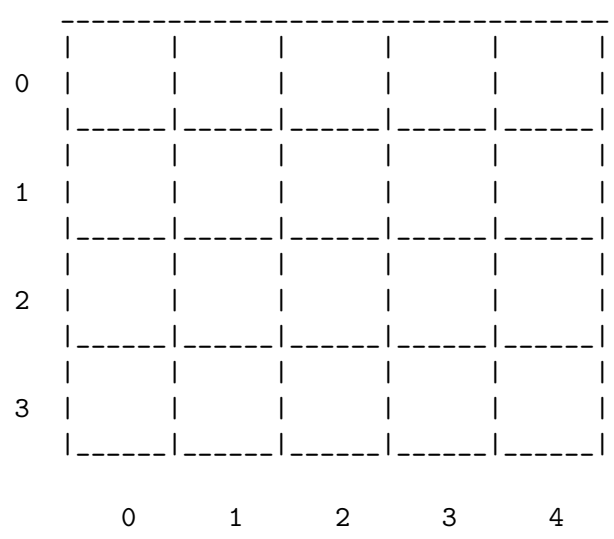
```
System.out.println(theMatrix[1][3]);
```

Typically, if you are viewing pictures of two-dimensional arrays, you see the indices of the first dimension used to select a row, and then the indices of the second dimension area used to select a column within that row. However, it is important to point out that, even though that is the "typical" way you tend to see examples, it is not necessary to think of things that way. All that matters is that you are consistent.

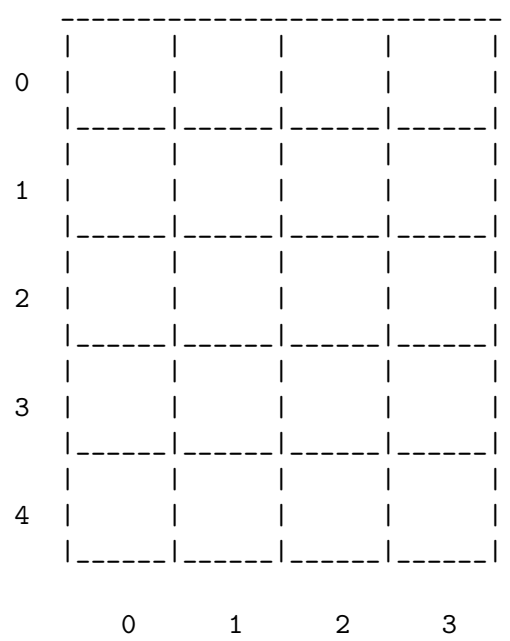
For example, go back to the earlier statement that created a two-dimensional array:

```
int[][] theMatrix = new int[4][5];
```

What is important about that statement, is that the *first dimension* is of size `4` (and thus has indices `0` through `3`) and the *second dimension* is of size `5` (and thus has indices `0` through `4`). Making sure you only use an index from `0` through `3` for the first dimension, and making sure you only use an `index` from `0` through `4` for the second dimension, is what is important. If you want to picture the first dimension as indexing the rows, then you'd get this picture (which you saw earlier):



But if you instead wanted to picture the first dimension as indexing the columns, you'd have this picture instead, and that would be fine too:



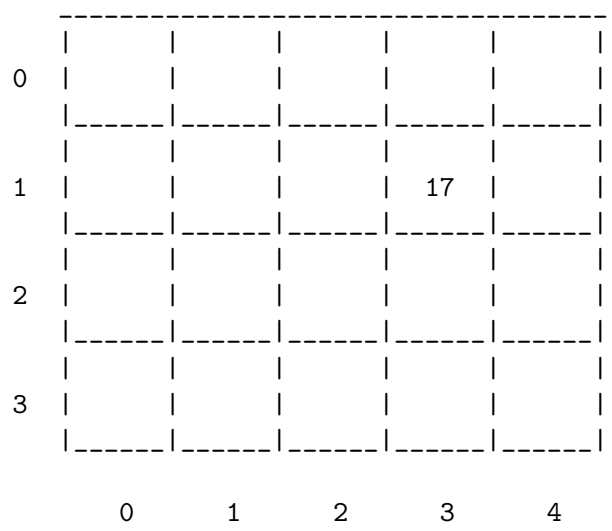
The important thing is that, whatever dimension – rows or columns – you decide will be your first dimension, that is *always* what must be your first dimension, for as long as the array exists. The code is only defined in terms of "first dimension" and "second dimension". It is *you* that decides whether defining the first dimension as indexing the "rows", or the "columns", makes more sense to *you*. So just as with any other definition, once you decide what "row" or "column" means to you, you need to be consistent.

For example, given the two-dimensional array you just created, if you try to make the following assignment:

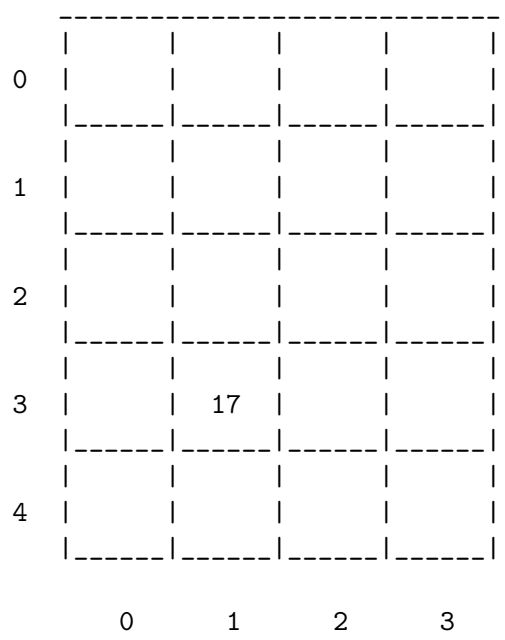
```
theMatrix[1][3] = 17;
```

then the cell you are accessing has index `1` in the first dimension, and index `3` in the second dimension. *That* is what is important, and relevant. Whatever you chose for the first dimension – rows or columns – you are accessing index `1` in that dimension, and whatever you chose for the second dimension – rows or columns – you are accessing index `3` in that dimension.

So, if you decided that the first dimension would index your rows, and thus that you have four rows, and five columns, then that assignment would select row `1` and column `3` and write a `17` into that cell, because the assignment selects index `1` from your first dimension, and *you* have decided that your first dimension indexes the "rows":



On the other hand, if you decided that the first dimension would index your columns, and thus that you have four columns and five rows, then that assignment would select *column* `1` and *row* `3`, and write a `17` into that cell, because the assignment selects index `1` from your first dimension, and *you* have decided that your first dimension indexes the "columns":



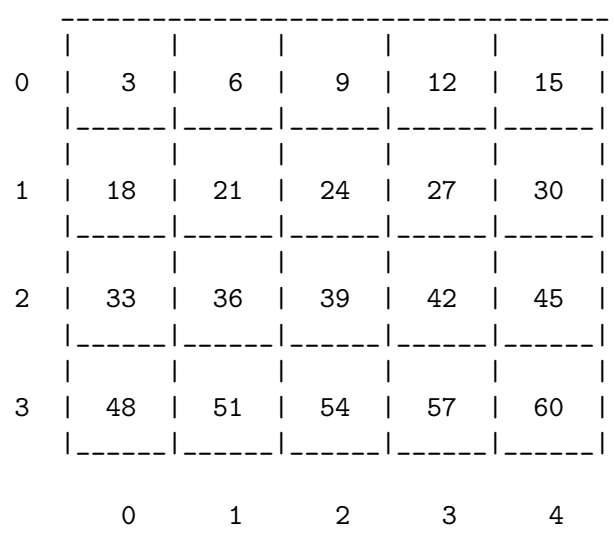
In *both* examples, the index of the first dimension is `1`. Whether that means "row `1`" or "column `1`" depends *entirely* on whether *you* decided the first dimension should index "rows" or "columns". It has nothing to do with the language itself. The language itself is only accessing index `1` in the first dimension, since the language only deals in concepts like "first dimension" and "second dimension", and does NOT deal in concepts like "rows" and "columns".

The use of `.length` in multi-dimensional arrays

Imagine we created a two-dimensional array as in our earlier examples:

```
int[][] theMatrix = new int[4][5];
```

Furthermore, imagine we have written values into all twenty cells, as follows:

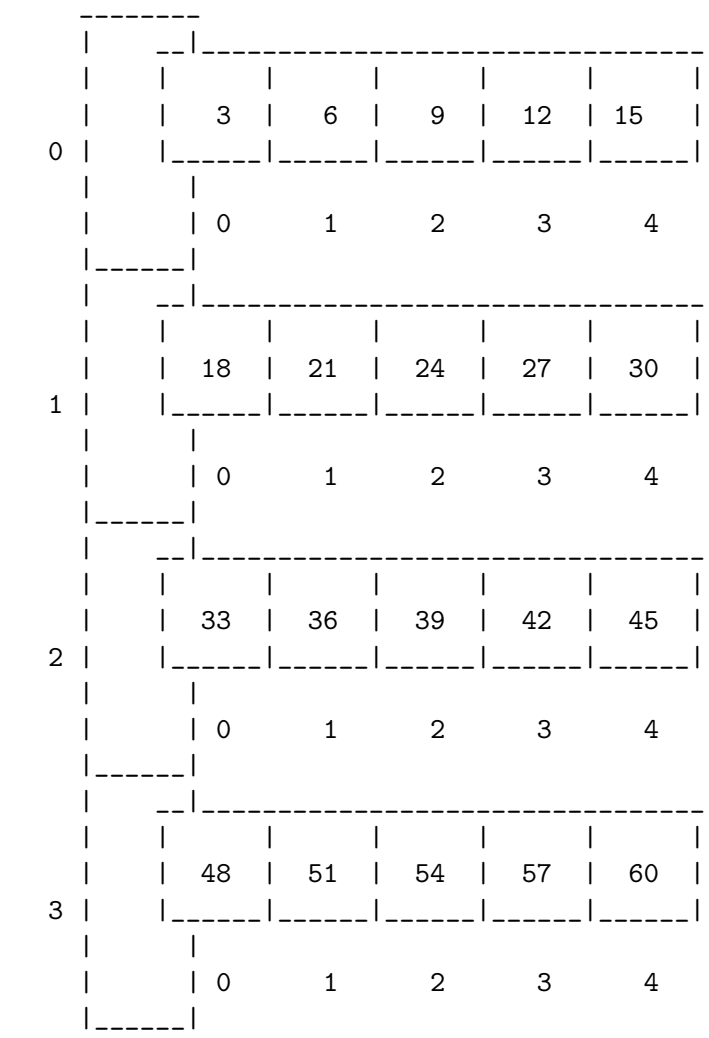


What value will the expression:

```
theMatrix.length}
```

produce? How do we obtain the length of the first dimension (which is `4`)? What code would we write to obtain the length of the first dimension (which is `4`)? What code would we write to obtain the length of the second dimension (which is `5`)?

The best way to answer these questions is with the picture on the following page, in which the first dimension indexes the rows, and the second dimension indexes the columns. It's a good way to think about a two-dimensional array. Imagine it as an array of arrays – where each row is a one-dimensional array holding the items in that row, and then there is another array holding all of those rows as its items. Look at the above picture, and then look at the picture on the next page, to see how we can view each row above as a one-dimensional array of size `5`, being stored in either cell `0`, `1`, `2`, or `3` of an array of size `4`.



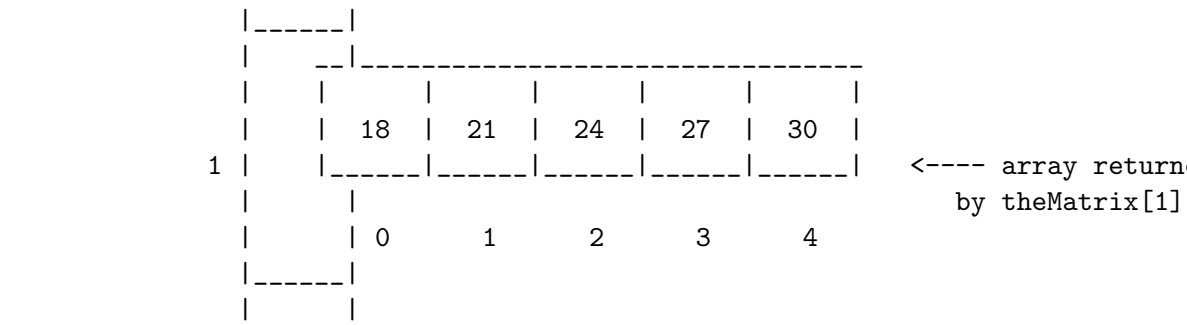
Given the above picture, imagine for a moment that the name of the two-dimensional array, `theMatrix`, is instead the name of the one-dimensional array of size `4` that is drawn vertically in the above picture. If that is the case, then we can consider an array access such as:

```
theMatrix[1][3]
```

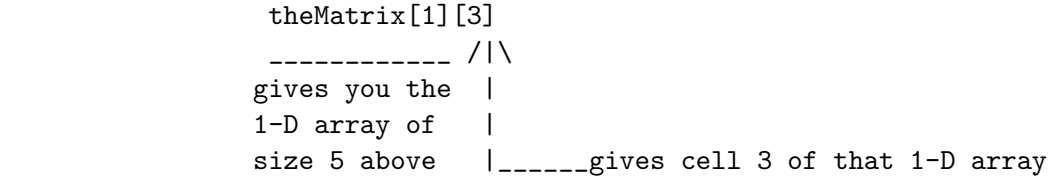
to be composed of two parts. The first part is selecting index `1` from the first dimension – i.e. accessing index `1` of the one-dimensional array of size `4` that is drawn vertically in the picture above:

```
theMatrix[1]
```

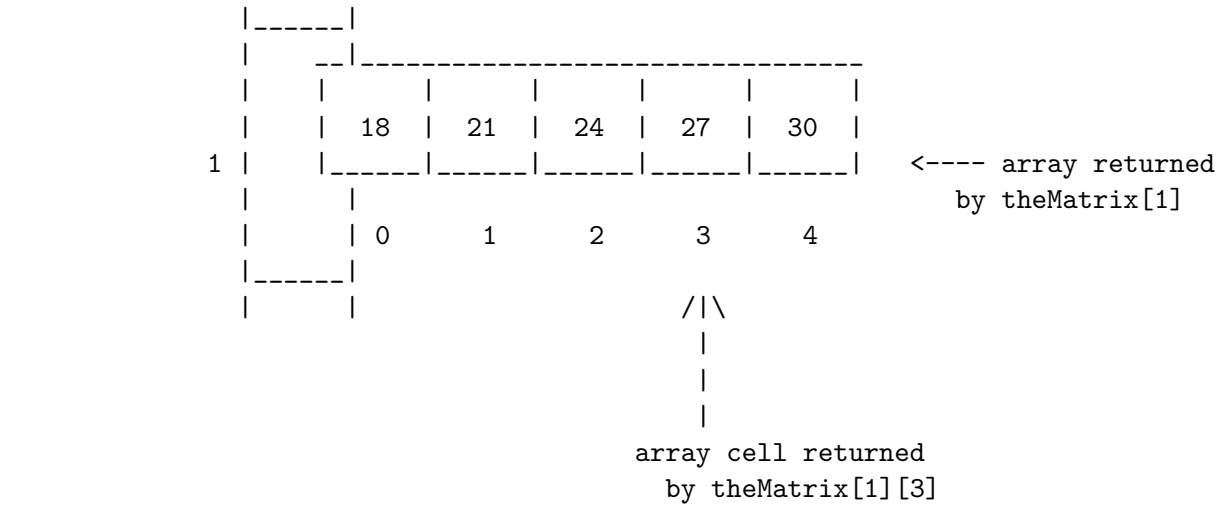
As we would expect for any one-dimensional array, this expression will return the item at cell `1` of the array drawn vertically in the picture above. And, what is at cell `1` of that array? Another array!



And then, the one-dimensional array we have obtained in this matter, has its cell with index `3` accessed.



giving you the cell that currently holds `27`:



That is, it sometimes helps to think of a two-dimensional array as an *array of arrays*. Think of the name of the two-dimensional array as the name of the vertical array in the previous pictures. That is, the name of the array is the "name of the first dimension", and then the cells of that first-dimension array, hold the actual rows representing the second dimension of the array.

That would mean that the expression `theMatrix.length` should evaluate to the length of that array that the name `theMatrix` refers to – namely, the vertical array in our pictures above, the first dimension of the array. And in fact, that is exactly what it does. The expression `theMatrix.length` will return the length of the *first dimension* of the array. In our examples above, the length of the first dimension is `4`, and so that is exactly what the expression `theMatrix.length` evaluates to – the value `4`.

So how can we get the length of the rows themselves? – i.e. the number of columns? – i.e. the length of the second dimension? Well, remember that the expression:

```
theMatrix[1]
```

in a sense gave us the particular row at index `1`, so that we could then access cell three of that array by adding a `[3]` to the end of that expression:

```
theMatrix[1][3]
```

So, instead of accessing the cell at index `3` of that row, why not try and obtain the length of the row instead? If we had the variable `arr` which was the name of an array, and we wanted to access cell `3`, we'd use:

```
arr[3]
```

and if we instead wanted the length of the array, we'd use:

```
arr.length
```

Well, instead of `arr` being the name of an array, we are now using:

```
theMatrix[1]
```

to access the one-dimensional array representing the row with index `1`, so why not use:

```
theMatrix[1].length
```

to get the length of that row? And, in fact, that is what we do! Since the following expressions:

```
theMatrix[0]
theMatrix[1]
theMatrix[2]
theMatrix[3]
```

would return the one-dimensional arrays representing the rows at index `0`, index `1`, index `0`, and index `3`, respectively...then the expressions:

```
theMatrix[0].length
theMatrix[1].length
theMatrix[2].length
theMatrix[3].length
```

will return the lengths of the one-dimensional arrays representing the rows at index `0`, index `1`, index `0`, and index `3`, respectively. And, of course, all those rows have a length of `5`, so each of the four expressions involving `.length` above, evaluates to `5`.

So, that is how you obtain the length of the first dimension and second dimension of some two-dimensional array `twoDarray`. The length of the first dimension is given by:

```
twoDarray.length
```

and the length of the second dimension would be given by

```
twoDarray[0].length
```

or by any similar expression, where in place of `0` you had some other index that was within the legal index range for the first dimension (an index between `0` and `3`, inclusive, in the four row, five column array of our earlier example).

And then likewise, for a three-dimensional array `threeDarray`, the following expressions:

```
threeDarray.length
threeDarray[0].length
threeDarray[0][0].length
```

would give you the lengths of the first, second, and third dimensions, respectively – due to much the same reasoning as we used earlier to explain the two-dimensional array example.

An example – initializing a two-dimensional array

Assume we have created a two-dimensional array, where rows are the first dimension and columns are the second dimension:

```
int[][] theMatrix = new int[4][5];  // array from previous examples
```

Furthermore, assume we would like to write the value 37 into every one of the twenty cells of this array. How could we do it?

Well, the key to this problem will be to use loops, since we want to repeat an action (writing 37 into an array cell) over and over again. To begin with, we note that if we are writing 37 into every array cell, that will involve writing 37 into every array cell in row 0, writing 37 into every array cell in row 2, and writing 37 into every array cell in row 3. Or in other words, we can begin our structuring of the solution, as follows:

```
for each row from 0 through 3
   write 37 into every cell of that row
```

which we can then translate into a for-loop:

```
for (int r = 0; r <= 3; r++)
   write 37 into every cell of row r
```

Now, if we want to write 37 into every cell of row r, well, that itself is a repetitive process. There are many columns in that row, and we are writing 37 into the cell for each column in that row. If we were doing that for a stand-alone one-dimensional array of size 5 named arr:

```
for every index in arr from 0 through index 4
   write 37 into the cell of arr at that index
```

and the way to do that for a one-dimensional array would be as follows:

```
// how to write 37 into every cell of some array arr with
//   indices 0 through 4:
for (int i = 0; i <= 4; i++)
   arr[i] = 37;
```

We saw code like that in the previous lecture packet, when talking about one-dimensional arrays.

And, the only difference between our situation and that one, is that rather than dealing with a stand-alone, one-dimensional array, we are trying to write 37 into indices 0 through 4 of a one-dimensional array that is *part of a two-dimensional array.* We are writing 37 into every cell in a particular row...but that isn't much different than writing 37 into every cell of a stand-alone one-dimensional array!

```
// given a row r, write 37 into every cell in that row
for all columns c from 0 through 4
   write 37 into the cell with row r and column c
```

or, once converted to a for-loop:

```
// assuming you know the index r of your row
for (int c = 0; c <= 4; c++)
   theMatrix[r][c] = 37;
```

That code will write `37` into every column of a given row `r` in our example array. Then, to do that for *every* row, we just use the above loop as the statement of our earlier loop:

```
for (int r = 0; r <= 3; r++)
   for (int c = 0; c <= 4; c++)
      theMatrix[r][c] = 37;
```

and we are done!

Remember that the entire statement for a loop (the entire body of the loop) runs from start to finish before the condition is checked again. And, in the case of a `for`-loop, the entire body of the loop is run before even the "alteration statement" (the `r++` or `c++` above) is run, let alone before the condition is checked. So, that means above, for the outer `for`-loop:

```
for (int r = 0; r <= 3; r++)
```

the statement for that loop – i.e. the inner `for`-loop – is run from start to finish before `r` is incremented or `r` is compared to `3` again.

This means that our progression of events is as follows:

```
   r is initialized to 0
   r is compared to 3
   enter body of loop
                        c is initialized to 0
                        c is compared to 4
                        enter body of loop
                                                write 37 into (0, 0)
                        c is incremented to 1
                        c is compared to 4
                        enter body of loop
                                                write 37 into (0, 1)
                        c is incremented to 2
                        c is compared to 4
                        enter body of loop
                                                write 37 into (0, 2)
                        c is incremented to 3
                        c is compared to 4
                        enter body of loop
                                                write 37 into (0, 3)
                        c is incremented to 4
                        c is compared to 4
                        enter body of loop
                                                write 37 into (0, 4)
                        c is incremented to 5
                        c is compared to 4
                        loop ends!
   r is incremented to 1
   r is compared to 3
   enter body of loop
                        c is initialized to 0
                        c is compared to 4
                        enter body of loop
                                                write 37 into (1, 0)
                        c is incremented to 1
                        c is compared to 4
                        enter body of loop
                                                write 37 into (1, 1)

(and so on...)
```

The inner loop runs from start to finish, writing `37` into every column of an entire row, before the row index (the index of the outer loop) is incremented to give us a new row.

This is the general structure for traversing a two-dimensional array – use two loops, one nested in the other, to traverse over every column in a row, for every row. If you wanted to do something other than write `37` into each cell, you'd follow the same pattern. For example, if we wanted to write the first twenty positive multiples of `3` into our array (as we had in an earlier picture in this packet), you'd move across the array cell by cell in the same manner, only setting up your next value to write into the array before you actually wrote into the array. That is, you would start out with the general form:

```
for (int r = 0; r <= 3; r++)
   for (int c = 0; c <= 4; c++)
      // whatever you want to do for each cell goes here
```

and then, the "whatever you want to do for each cell" is, "figure out the next multiple of `3` and write it into that cell". The next multiple of `3`, will be the previous multiple of `3`, plus `3`. (For example, given `21`, we add `3` to get `24`, then add another `3` to get `27`, and so on.) So, let's have a counter keep track of the multiple of `3` that we care about at the moment:

```
int counter = 0;
for (int r = 0; r <= 3; r++)
   for (int c = 0; c <= 4; c++)
   {
      counter = counter + 3;
      theMatrix[r][c] = counter;
   }
```

Note that we needed to supply curly braces to make a compound statement, since now we are trying to run two statements – `counter = counter + 3;`, and the array cell assignment – as the body of the inner `for`-loop. However, the inner `for`-loop itself is still one statement, so the *outer* `for`-loop does not need curly braces around its statement (i.e we don't need curly braces around the inner `for`-loop itself). That said, sometimes it helps make the code easier to read if we put them in, even if they aren't necessary. So, the above code could also be written as:

```
int counter = 0;
for (int r = 0; r <= 3; r++)
{
   for (int c = 0; c <= 4; c++)
   {
      counter = counter + 3;
      theMatrix[r][c] = counter;
   }
}
```

In either case, the progression of events would be as follows:

```
counter is declared and initialized to 0
r is initialized to 0
r is compared to 3
enter body of loop
                    c is initialized to 0
                    c is compared to 4
                    enter body of loop     counter is incremented by 3
                                           write 3 into (0, 0)
                    c is incremented to 1
                    c is compared to 4
                    enter body of loop
                                           counter is incremented by 3
                                           write 6 into (0, 1)
                    c is incremented to 2
                    c is compared to 4
                    enter body of loop
                                           counter is incremented by 3
                                           write 9 into (0, 2)
                    c is incremented to 3
                    c is compared to 4
                    enter body of loop
                                           counter is incremented by 3
                                           write 12 into (0, 3)
                    c is incremented to 4
                    c is compared to 4
                    enter body of loop
                                           counter is incremented by 3
                                           write 15 into (0, 4)
                    c is incremented to 5
                    c is compared to 4
                    loop ends!
r is incremented to 1
r is compared to 3
enter body of loop
                    c is initialized to 0
                    c is compared to 4
                    enter body of loop
                                           counter is incremented by 3
                                           write 18 into (1, 0)
                    c is incremented to 1
                    c is compared to 4
                    enter body of loop
                                           counter is incremented by 3
                                           write 21 into (1, 1)
```

...and so on. Note that when we start writing values into the row with index `1`, we start with `18`. The variable `counter` was declared outside the outer `for`-loop, so the changes we make to it inside

the loops are permanent. Every time we add `3` to the current value of `counter`, we are adding `3` to the most recent value of `counter`, which is the value it held after the most recent assignment to `counter` – which is also the value we most recently wrote into the array. Now, if instead, we wanted to write `3`, `6`, `9`, `12`, and `15` into *each* of the four rows of the array, then we would need to reassign `counter` to zero just before starting the inner loop each time. That way, the first cell of each row would always have `3` assigned to it, because just before that assignment, we had run the line `counter = counter + 3;`, and if `counter` was `0` before that line was run, then after it was run `counter` would be `3`. The full example for this would be as follows:

```
int counter;
for (int r = 0; r <= 3; r++)
{
   counter = 0;
   for (int c = 0; c <= 4; c++)
   {
      counter = counter + 3;
      theMatrix[r][c] = counter;
   }
}
```

That code would write `3`, `6`, `9`, `12`, and `15` into *each* row of the array.

Three additional issues with arrays

- One interesting and useful property of arrays, is that it takes the same amount of time to access an array cell no matter what the index is. For example, if you have a one-dimensional array `arr` of size `10000`, the first and last cells are obtained via the expressions `arr[0]` and `arr[9999]`, respectively. You might think it would take longer to access `arr[9999]`, since it's further down the array, but actually, whether the index is `0` or `9999` or `4999` makes no difference, with respect to how long it takes the computer to access the array cell. Now, the reasons this is true are beyond the scope of this course – you need to learn a bit more about computer hardware before you can fully understand *why* this is true. But at any rate, it is indeed true. This means that when you write code using arrays, you don't need to be shy about using cells with higher indices – writing a value into `arr[9999]` rather than `arr[0]` is NOT going to "slow down" your program. Treat each array cell, no matter the index, as if is just as easily accessible as any other array cell.

- Once an array is created, you *cannot* change its size. So, for example, the following would not work:

```
int[] x = new int[5];
     .
     .
     .
x.length = 10;
```

  In fact, that code would not even compile! If you need an array of size `10` at that point in your program, you will need to create a second array – there is no way to expand the size of the first array.

```
// this is how you would have to do this
int[] x = new int[5];
     .
     .
     .
int[] y = new int[10];
```

- The assignment operator does not work for arrays the same way it does for single variables. That is, if you had created two arrays of the same size:

```
int[] x = new int[6];
int[] y = new int[6];
```

  and then wrote different values into those array cells, if you later want to make the two arrays hold the same values, you should NOT do this:

```
x = y;
```

  Instead, you need to copy the values cell by cell, as in the following code:

```
// this code assumes both arrays have at least 6 cells, indexed
//   0 through 5
for (int i = 0; i <= 5; i++)
   x[i] = y[i];
```

There are reasons for this that we will discuss soon; for now, just know that assignment of an entire array to another entire array needs to be done one array cell at a time, not all at once.