Our Flounder

Computer Science Correspondence School

discere domi

Lecture 22

CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Multi-Dimension Arrays

Due: **Monday, May 11, 2020, at start of class**
Points: **None**

Start of Today's Lecture

If you had this array declaration in your Javascript program:

```
var nameArray;
nameArray = [ "Liam", "Elaine", "Anne", "Diane" ];
```

the array could be thought of as a row of values:

element 0	***element 1***	***element 2***	***element 3***
Liam	Elaine	Anne	Diane

or as a column of values

element 0	Liam
element 1	Elaine
element 2	Anne
element 3	Diane

Each element in the array is associated with a single value and that value can be accessed only by using a subscript. As you have seen, arrays are very useful and, once you have practiced a bit, convenient and easy to access.

You have also used associated arrays - two arrays that have the same number of elements and where the i_{th} element of each array is related. For example, here is a pair of associated arrays that, taken together, store a person's name:

```
var firstName;
var lastName;

firstName = [ "Susan", "George", "Carol",     "Leo",   "Alice" ];
lastName  = [ "Hall",  "Cooper", "Arkwright", "Green", "Adams" ];
```

These arrays could be visualized this way. The element number is shown in the leftmost column.

	firstName	**lastName**
0	Susan	Hall
1	George	Cooper
2	Carol	Arkwright
3	Leo	Green
4	Alice	Adams

In order to access a person's full name, it would be necessary to do something like this:

```
firstName[i] + " " + lastName[i]
```

Each element in an array has a value and, in the arrays shown above, that value was a *String*. However, the value of an element could be anything including a reference to another array. A *multi-dimension array* is an array in which the element values are references to other arrays. Multidimension means more than one. For example, you could have a two dimension array, or a three dimension array. A one dimension array is not a multidimension array. It is just called an array. It is the most commonly used type and the type you have been using so far.

While arrays can be of any dimension, the two dimension array is most commonly used multidimension array because most problems require only one or two dimension arrays. While a single dimension array can be visualized as a row or column, a two dimension arrays can be visualized as a grid, or checkerboard, or a spreadsheet, if you have used a spreadsheet program. The associated arrays shown above could be visualized as a two dimension array:

Susan	Hall
George	Cooper
Carol	Arkwright
Leo	Green
Alice	Adams

This grid is made up of *rows* and *columns*:

	columns	
	Susan	Hall
r	George	Cooper
o	Carol	Arkwright
w	Leo	Green
s	Alice	Adams

In order to access the first name of a person or the last name of a person, you need to specify two things:

- the *row* number of the person, and
- the *column* number of the person

Taken together, the row number and the column number intersect on a single element in the grid. The row number and column number are subscript values and, for the above array, are shown below in the leftmost column (row number) and top row (column number)

	0	**1**
0	Susan	Hall
1	George	Cooper
2	Carol	Arkwright
3	Leo	Green
4	Alice	Adams

If we assume that the variable that is referencing the two dimension array is named *personName*, then the *personName* two dimension array has five rows (zero through four) and two columns (zero and one). In order to access either the first name or the last name, you must use two subscripts. The row subscript must be between zero and four and the column subscript must be between zero and one. Here are some examples

`personName [2][0]` is a reference to *Carol*
`personName [2][1]` is a reference to *Arkwrigth*
`personName [4][0]` is a reference to *Alice*
`personName [4][1]` is a reference to *Adams*

If you wanted to reference a person's full name you could do this:

```
personName [3][0] + ‟ ‟ + personName [3][1]
```

which would produce:

```
Leo Green
```

In Javascript, you can declare an array variable and give its elements initial values without using a loop. You have done this many times with single dimension arrays and you can do it with multiple dimension arrays, too. Here is how the personName two dimension array could be created and initialized.

```
        var personName;
        personName = [
                      ["Susan",  "Hall"],       // row zero
                      ["George", "Cooper"],     // row one
                      ["Carol",  "Arkwrigth"],  // row two
                      ["Leo",    "Green"],      // row three
                      ["Alice",  "Adams"]       // row four
                    ];
```

As is shown above, each row of the personName array contains a two element single dimension array. In this kind of creation/initialization, the number of dimensions in the array is identified by the maximum depth of nested `[]` in the declaration. Here the [ ] are nested two deep and, therefore, it is a two dimension array.

Here is another example that shows the creation and initialization of a two dimension array of numbers. The *count* array has three rows and each row has four columns.

```
                  var count;
                  count = [
                            [1, 2, 3, 4],      // row zero
                            [5, 6, 7, 8],      // row one
                            [9, 10, 11, 12]    // row two
                          ];
```

The *length* property of a single dimension array tells you the number of elements in the array. If the array is a two dimension array, the *length* property tells you how many rows are in the array. The *count* array shownn above is a two dimension array, so

- `count.length` is three meaning that there are three rows in this two dimension array
- `count[i].length` is four because there are four columns in row *i* of the array

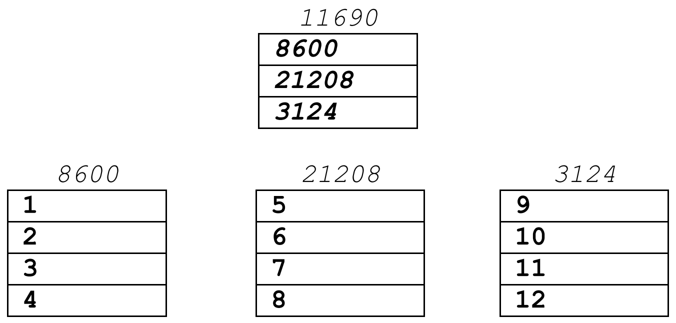
In order to access the lowest level element (*component* and *element* are synonyms) of an array, you need to use a subscript for each dimension. If you are using a single dimension array, you need one subscript to access the lowest level element. If you are using a two dimension array, you need to use two subscripts to access the lowest level element. If you are using a three dimension array, you need to use three subscripts to access the lowest level element, etc. In a two dimension array, the first subscript identifies the row number and the second subscript identifies the column within that row.

The above *count* array has three rows and four columns. Many programming languages allocate arrays (including multidimensional arrays) as a single block of memory and use a formula to identify the memory location of each of the lowest level elements. In these languages, all two dimension arrays have a rectangular shape: each row has the same number of colums. For example here is how the count array would appear in memory: just a contiguous number of bytes:

1	2	3	4	5	6	7	8	9	10	11	12

Javascript uses a different technique. Javascript, and most modern languages, use an *array of arrays* approach when storing multidimensional arrays. In this approach, the values in each row (these are the values of the columns for a row) of the array are placed in a single block of memory and an additional block is used to identify the location of the rows.

Here is how the above *count* array would look when placed in memory using the array of arrays approach: The values in italics are memory addresses (memory reference). Assume that the value stored in the *count* variable is the memory address *11690*.



The memory addresses are shown only as an example. There is no way to know which memory addresses will be used by the system when the array is created.

Although it is not stored in memory in this way, it is easier to think of a two dimensional array as a grid, or spreadsheet. Here is how the above 3 by 4 table would look if pictured as a grid:

1	2	3	4
5	6	7	8
9	10	11	12

To access the value 7 from this table, you would need to use the subscripts `[1][2]`. This is interpreted as column 2 of row 1 (remember, things are numbered from zero in Javascript). Here are the subscripts that would access the corresponding values in the above table.

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

Because of the *array of arrays* approach that Javascript takes with multidimensional arrays, each subarray is an array and has its own *length* property. It is always the best programming approach to use this *length* property rather than either hard coding the size in your program or making the assumption that every row has the same number of columns. Indeed, each row may have a different number of columns in the *array of arrays* approach.

Here is an example of creating a three row by row column array:

```
var col;
var count;  // the name of the array
var row;
var x;

count = new Array ( 3 );  // three rows
for (row =0; row<count.length; row++)
{
 // each row will have four columns;
 count[row] = new Array(4);
}
```

The array has been created but the values in the array have not been intitalized. All have the value *undefined*. This code will initialize the array to have the values shown in the above example:

```
x = 1;
for ( row=0; row<count.length; row++ )
{
 for ( col=0; col<count[row].length; col++)
 {
  count[row][col] = x;
  x = x + 1;
 } // for ( col ...
}  // for ( row ...
```

Notice how two subscripts (one identifying the row number and one identifying the column number in that row) are needed to access an element in the two dimension array.

Here is one way to sum the elements of the above array.

```
var col;
var row;
var sum;
sum = 0;
for(row=0; row<3; row++)
{
 for(col=0; col<4; col++){ sum = sum + count[row][col]; }
}
```

This is *not* the best way to write this code. If you changed the size of the array in the declaration, the code would no longer work. Here is a much better way that incorporates the use of the *length* variable.

```
var col;
var row;
var sum;
sum = 0;
for(row=0; row<count.length; row++)
{
  for(col=0; col<count[row].length; col++)
  {
   sum = sum + count[row][col];
  }
}
```

Notice that this accommodates each row having a different number of columns (*length*). Arrays that have the same number of columns in each row are *rectangular*. Often i and j are used as row subscripts and column subcripts. Here is a function that, when passed a two dimension array, will return the sum of the elements in that array. Notice the use of *i* and *j* as subscript variables.

```
function sumTwoDimensionArray ( array2d )
{
 var i;
 var j;
 var sum;
 sum = 0;
 for (i=0; i<array2d.length; i++)
 {
  for (j=0; j<array2d[i].length; j++)
  {
   sum = sum + array2d[i][j];
  }
 }
 return sum;
}
```

Remember that, if you have a two dimension array named *x*, then *x.length* tells you how many rows are in the array and *x[i].length* tells you how many columns are in row *i* of the array. Always use the *length* property when manipulating arrays.

For the following problems, the functions you write should not modify the contents of the parameter array unless specifically told to do so,.

Problem 1:
Write a function with this function header:

```
function isRectangular (twoDimenstionArray)
```

The function is passed a reference to a two dimension array. You can assume that the reference is never *null*. The function returns *Boolean true* if every row in the array has the same number of columns, otherwise, it returns *Boolean false*.

Here is one way of writing this function:

```
function isRectangular ( twoDimensionArray )
{
 var i;
 var j;
 var result;

 result = true;
 for(i=1; i<twoDimensionArray.length;i++)
 {
  result = result &&
        ( twoDimensionArray[i].length === twoDimensionArray[i-1].length );
 }
 return result;
}
```

Problem 2:
Write a function with this function header:

```
function concatenate (twoDimenstionArray)
```

The function is passed a reference to a two dimension array. The elements of the array could be Strings or numbers or anything. The function returns a String that is the result of concatenating all the elements of the array together.

Problem 3:
Write a function with this function header:

```
function create (numberOfRows, numberOfColumns, intialValue)
```

The function is passed an integer number of rows and and integer number of columns. Each of these is greater than zero. The function returns a rectangular array with that number of rows and than number of columns. The elements of the array are intialized to *initialValue*.