CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Introduction to Arrays

Due: **Tuesday, April 7, 2020**
Points: **none**

Start of Today's Lecture

Each variable that you declare in a program holds one value. That value can be a piece of data (for example, it could be the number *6*) or a reference to memory such as, *214596* that might point to a *String* that contains the characters *Bob*. In the latter case, the machine uses the value of the variable to access the true data that is associated with it. For example, when you use the *charAt(k)* function on a *String*, the value stored in the *String* variable is used to access that part of the memory where the characters of the *String* are stored and then the character that is *k* characters from the start of that piece of memory is returned[1].

Here is a function that will cause a *window.alert* to appear that contains the character number and associated character for each character in a String.

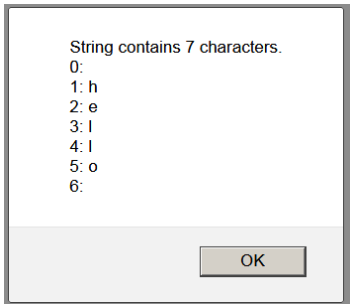
```
function stringDisplay ( someString )
{
 var i;
 var result;

 result = "String contains " + someString.length + " characters.";
 i = 0;
 while ( i < someString.length )
 {
  result = result + "\n" + i + ": " + someString.charAt ( i );
  i      = i + 1;
 }

 window.alert ( result );
}  // stringDisplay
```

If you had a *window.onload* function that looked like this:

```
window.onload = function()
{
 stringDisplay ( " hello " );
};
```

[1] This is a conceptual view of what is happening. The actual implementation is left to the specific Javascript interpreter.

The browser would display this dialog box when the page was loaded:



While *Strings* are useful, they are limited. A *String* can store only characters. It cannot store numeric values or more complex things such as other *Strings*. There is a more general purpose data structure that can do this, though. This structure is called an *array*. The array can hold zero or more of any type of value. These values are called *elements* or *components*. *Elements* and *components* are the same thing.

Every programming language implements *arrays* and most implement them in the same way. Javascript has an unusual approach to some array features but we will not use these since their applicability is limited to the Javascript language. We will use only the array features of Javascript that are common across most languages and limit our programming style to those features and restrictions. This will help make an easier transition from writing programs in Javascript to writing them in other languages.

Arrays are stored in a manner similar to *Strings.* When you store a reference to an array in a variable, it is very similar to storing a reference to a *String* in a variable. The value of an array variable is a memory address. The components of the array are stored beginning at that memory address[2] just like the value of a *String* variable is a memory reference and its associated characters are stored beginning at that memory address. While the contents of a *String* are limited to be only characters, the contents of an array are elements and can be of any data type: *String*, number, etc.

Strings are created by using a non-numeric literal – some text inside quotation marks or apostrophes – or through concatenation or using one of the *String* methods. As you have discovered, once a *String* has been created, its contents and length cannot be changed. For example, the *charAt(i)* function cannot appear on the left side of the assignment operator which means that it cannot be used to change a character in the *String*. That function can only be used to copy a character from the *String*.

Arrays are a little more flexible but, like *Strings*, once they are created, their length (the number of elements in the array) cannot change. The length is fixed for the entire existence of the array[3]. However, any of the components (elements) of the array can appear on the left side of an assignment operator and, therefore, you can change the value of any of the components in the array. You just can't make the array have more or fewer components.

[2] Javascript has a different way of doing this but it appears to the programmer that it uses this common approach and we will pretend that it implements it this way.
[3] Once again, this is different in Javascript but we will use this common approach when we write our Javascript programs.

Finding out the number of characters in a *String* or the number of components in an array is done in the same way: by using the *length* property. Just like the *String*, the components of the array are numbered starting from zero and continue through *length-1*. To retrieve (access) the value of a component of an array, you must use a *subscript*. The *subscript* is analogous to the actual parameter value used with the *String charAt* method.

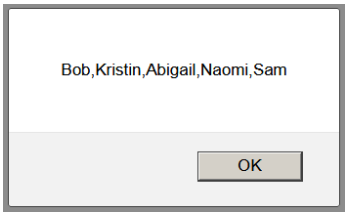
A *subscript* is an integer value that is greater than or equal to zero and less than the *length* of the array. This value can be a variable, a constant, or some numeric expression that evaluates to an integer in the correct range. The subscript must appear inside square braces and the square braces must immediately follow the name of the variable that contains the array reference. For example, if *x* is the variable that contains the array reference, then *x[j]* will access component number *j* of the array. If this array reference is to the left of an assignment operator, the value associated with component number *j* will be changed. If it does not appear to the left of the assignment operator, the value of the associated with component *j* will be accessed.

Once an array has been created, the value of each component of the array *must* be initialized. If you create an array whose *length* is fifty, then the array contains fifty components but the value of each component is *undefined*. Most programming languages have syntax that lets you combine the creation and initialization but you will find that its use is limited to those cases where, in advance, you know both the size and the contents of the array. Here is an example of an *onload* function that creates two arrays using this kind of syntax:

```
window.onload = function()
{
 var birthYear;
 var name;

 birthYear = [ 2000, 1999, 2001 ];
 name      = [ "Bob", "Kristin", "Abigail", "Naomi", "Sam" ];

};
```

When the above syntax is used to create an array, the interpreter counts the number of elements, creates the array using that count, and then initializes each element using the value specified.

When using arrays in programs, it is often useful to display the contents of the array. This can be done by using the array variable name in a *window.alert* statement. The dialog box will contain the values in the array each separated by a comma. This is what would appear if you did *window.alert (name)*:



It is a convenient way to see the array contents but it is not always easy to determine the element number associated with each value and, for larger arrays, it gets confusing.

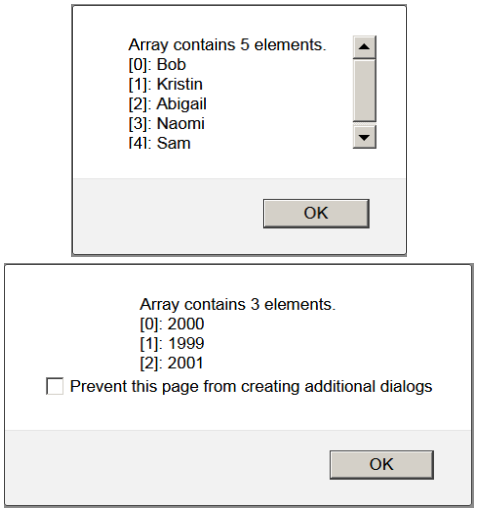
The following function displays the contents of the array in a more clear way and I recommend its use. Compare the function shown here with the *stringDisplay* function shown earlier. Notice the similarities.

```
function arrayDisplay ( someArray )
{
 var i;
 var result;

 result = "Array contains " + someArray.length + " elements.";
 i = 0;
 while ( i < someArray.length )
 {
  result = result + "\n[" + i + "]: " + someArray[i];
  i      = i + 1;
 }

 window.alert ( result );
}  // arrayDisplay
```

Modify the *onload* function to this:

```
window.onload = function()
{
 var birthYear;
 var name;

 birthYear = [ 2000, 1999, 2001 ];
 name      = [ "Bob", "Kristin", "Abigail", "Naomi", "Sam" ];

 arrayDisplay ( name );
 arrayDisplay ( birthYear );

};
```

When the page is opened in the browser, these two dialog boxes would be displayed.



This shortcut syntax is convenient but is only useful when the programmer knows exactly how big the array should be and exactly what the value of each element should be. Unfortunately, the most common case is that you know neither the size nor the exact value of each element of the array until your program begins executing. For these cases, the programmer needs to implement this approach:

- doing some kind of count of the things to be stored in the array
- creating the array using this count, and then

- initializing each component of the array to the desired value.

I refer to this process as: **count, create, populate** and it will be the process you use most often when dealing with arrays.

Example 1
Create the function with the following header:

```
function getDayName ( dayNumber )
```

The function is passed an integer whose value should be between one and seven. The function returns a *String* containing the name of the day of the week that corresponds to the number For example one represents *Monday* and seven represents *Sunday*. If the day number is not valid, the function should return a zero length *String*.

When using arrays, it is very important not to exceed the boundaries of the array. That is, the subscript used to access the array must always be between zero and one less than the array length. This is exactly the same rule you have been using when accessing characters in a *String*. For this function, the parameter value is between one and seven (one based counting) but the subscript value must be between zero and six (zero based counting). This is a common situation in programming.

Since there are only seven days in the week, it is possible to write this function without using arrays:

```
function getDayName ( dayNumber )
{
 var result;

 result = '';
 if(dayNumber === 1) result = 'Monday':
 if(dayNumber === 2) result = 'Tuesday:
 if(dayNumber === 3) result = 'Wednesday:
 if(dayNumber === 4) result = 'Thursday:
 if(dayNumber === 5) result = 'Friday:
 if(dayNumber === 6) result = 'Saturday:
 if(dayNumber === 7) result = 'Sunday:
 return result;
}
```

But it is not very elegant and works as well as it does only because there are so few days in the week. The array approach can make use of the shortcut syntax for creating and populating the array because we already know both that there are always seven days in the week and we know the names associated with each of those days. Therefore there is no need to take the *count-create-populate* approach. Here is one approach to writing this function using arrays:

```
function getDayName ( dayNumber )
{
var dayName;
var result;

dayName = ['Sunday', 'Monday', 'Tuesday', 'Wednesday',
           'Thursday', 'Friday', 'Saturday'];

result = '';
if(dayNumber >= 1 && dayNumber <= dayName.length)
{
 result = dayName[dayNumber-1];
}

return result;
```

Notice how the shortcut syntax uses the square braces to both create and populate the array. In this case, the array will contain *Strings*, so each item listed between the braces is a *String*. Also notice how the comma is used to separate the element values. This technique can be used with numbers as shown above; just list the numbers and use the comma as a separator.

The *if* statement checks to ensure that the parameter (a one based value) is in the correct range. You never want to access outside the boundaries of the array. When setting the value of the variable *result*, the parameter value must be changed from one based to zero based. That is why one is subtracted from *dayNumber* inside the square braces. Recall that the value inside the square braces is called the *subscript*.

You might wonder why *dayName.length* was used instead of the integer value *seven*. It is a very good programming practice to always use the *length* property of the array (just as you have done with *Strings*) and never use some specific value. The reason for this is that, should you change the size of the array at some later date, your program will still work correctly. Always use the *length* property.

Example 2

Create the function with the following header:

```
function getStringLengths ( stringArray )
```

The function is passed an array that contains *Strings*. The function returns an array that contains integer values. The value of each component of the returned array is the number of characters in the corresponding component in the parameter array.

This is a typical situation. A function is passed an array that could contain any number of elements and must return an array that, also, could contain any number of elements. For some problems, the number of elements in the two arrays may not be the same. For this problem, the number of elements in the returned array is the same as the number of elements in the parameter array because each *String* in the parameter array has a length that must be stored in the array that is returned by the function. This tells you how big to make the array that the function will return.

The function requires a loop (as will almost all array processing functions) to populate the components of the returned array. When creating an array whose length we know but whose content we don't know, we will use this syntax:

```
variableName = new Array( n );
```

where *n* is an integer greater than or equal to zero. This syntax will create an array that has *n* components and the value of each of these components will be set to *undefined*. Here is how the function might look:

```
function getStringLengths (stringArray)
var i;
var result;

// create the array to be returned. It will be the
// same size as the parameter array
result = new Array(stringArray.length);

// populate the array
i = 0;
while(i < result.length)
{
 result[i] = someArray[i].length;
 i = i + 1;
}

return result;
```

The statement *someArray[i]* is a reference to the *String* at element *i* the array. Using its.*length* property accesses the number of characters in that *String*.

Example 3

Create the function with the following header:

```
function getOddValues ( array )
```

The function is passed an array that contains integers. The function returns an array that contains only the odd integers in the parameter array.

In this situation, you know neither how large to make the returned array nor what its exact contents are. This is the general case and one which will come up quite often. In these situations, we must use the *count-create-populate* approach. Implement and test this function.

```
function getOddValues (array)
var count;
var i;
var result;
// the variable named "result" is the variable that will hold a
// reference to the array that will be created by this function.

//  Step 1
//  determine the size of the returned array by counting
count = 0;
i = 0;
while(i<array.length)
{
 if(array[i] %2 === 1) { count = count + 1; }
 i = i + 1;
}  // while

//  Step 2
//  create the array using the count
result = new Array(count);

//  Step 3
//  store a value in each element of the array
count = 0;
i = 0;
while(i<array.length)
{
 if(array[i] %2 === 1)
 {
  result[count] = array[i];
  count         = count + 1;
 }
 i = i + 1;
}  // while

return result;
```

Warning

You must always exercise caution when a function has an array formal parameter. While the value of the parameter (the memory address that points to the array) can be modified in the function without affecting the actual parameter, any changes you make to the values of the components of the formal parameter array *will* be made to the actual parameter array because they are the same array.

Exercises

Unless the problem specifically requires that you modify the parameter array, your function must never have a reference to a component of the formal parameter array appear to the left of an assignment operator.

Test each function thoroughly by displaying the return value of the function on the monitor using the *arrayDisplay* function written earlier.

Exercise A

Create the function with the following header:

```
function copy ( array )
```

The function is passed an array whose contents are unknown – they could be *Strings*, they could be numbers, they could be anything. The function returns an array that is the same size and contains the same contents as the passed array. Just returning a reference to the passed array will not work. You must duplicate the array. This means you must create a new array and copy each of the elements from the passed array into the new array and then return the variable that contains a reference to this new array

Exercise B

Create the function with the following header:

```
function letterCount ( someString )
```

The function is passed a *String* and returns an array. The function returns an array of length two. The first component of the returned array contains the number of vowels (in either upper or lower case) in the parameter. The second component contains the number of non-vowels in the parameter.

Exercise C

Create the function with the following header:

```
function sum (array)
```

The function is passed an array of numbers. It does not return an array. It returns a number which is the sum of the values in the parameter array.

Exercise D

Create the function with the following header:

```
function arrayToString ( array )
```

The function is passed an array of unknown contents. It does not return an array. It returns a *String* which is the result of concatenating together each component of the parameter array. Remember that the parameter array might not contain *Strings*, it might contain numbers and that your function cannot change the contents of any component of the parameter array.

Exercise E

Create the function with the following header:

```
function getStringsOfLength( anArray, numberOfCharacters )
```

The function is passed two parameters: an array containing *Strings* (*anArray*) and an integer (*numberOfCharacters*). The function returns an array whose contents are the *Strings* in the parameter array that have the same number of characters as the value of the parameter integer. For example, if the parameter integer had the value seven, the returned array would contain copies of all the *Strings* in the parameter array that contained exactly seven characters.

Exercise F

Create the function with the following header:

```
function stringToArray (data)
```

The function is passed a *String*. It returns an array in which the element *j* of the array is a *String* that contains only the character *j* of the parameter.