Computer Science Correspondence School

discere domi

Lecture 10

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

Due: Wednesday, April 15, 2020
Points: none

Solutions to Problems from Lecture 9

Here are some sample solutions to the array exercises you did last time. Your solutions may differ from mine, so study mine to see other ways to do the functions. There are almost always several algorithms that can be used to solve a problem. As you develop more algorithms, you will develop the ability to see these alternate solutions. Only write code that you understand. If you are writing code you don't understand, it becomes magic, not logic.

Exercise A

Create the function with the following header:

```
function addOrConcatenate ( array )
```

The formal parameter array either contains all *Strings* or all numbers and it contains at least one component (*i.e.*, its *length* property is greater than zero). If the array contains all numbers, the function returns their sum. If the array contains all *Strings*, the function returns all of the array elements concatenated together into one *String*. You may not use any Javascript that has not been introduced in class. For example, you may not use the *instanceof* operator. The solution is easy but determining what the solution is might take some time. Take some time and think about it. Test the function using an array of numbers and an array of *Strings*.

```
function addOrConcatenate ( array )
{
 var i;
 var result;

 if ( (array[0]+"") === array[0] )
  result = "";
  else
  result = 0;

 i = 0;
 while ( i < array.length )
 {
  result = result + array [ i ];
  i = i + 1;
 }
 return result;
}
```

The function takes advantage of the overloaded operator: plus sign. Recall that if that operator is in a *String* context, it does concatenation. If that operator is in a numeric context, it does addition. The *if* statement at the beginning of the function tests element zero of the array. It

compares element zero to the *String* version of element zero. If the two are equal, the function assumes that the array contains *Strings*. If the two are not equal, the function assumes the array contains numbers. Depending on the outcome of the test, the function sets the *result* variable to a zero length *String* (this will cause the overloaded operator to do concatenation) or to numeric zero (this will cause the overloaded operator to do addition).

Exercise B

Create the function with the following header:

```
function deleteElementZero ( array )
```

The function is passed an array and returns a newly created array. The array that is returned by the function has all the elements of the parameter array with the exception of element zero. For example, the parameter array contained these elements: *5, 9, 3 4* then the array returned by the function would contain only these elements: *9, 3, 4*. So, the *length* of the array returned by the function is one less than the *length* of the parameter array. If the parameter array *length* is zero, then the function should return a newly created array whose *length* is also zero.

```
function deleteElementZero ( array )
{
 var i;
 var result;

 if ( array.length > 0 )
  result = new Array ( array.length - 1 );
  else
  result = new Array ( 0 );
 i = 1;
 while ( i < array.length )
 {
  result [ i-1 ] = array [ i ];
  i = i + 1;
 }
 return result;
}
```

The function must be written to delete the first element of the array only if the array has a first element. That is what the *if* statement determines. If the array has no elements, a zero length array will be returned. If it has *n>0* elements, an array of size *n-1* will be returned. This function could also be written using a *Math.max* statement as shown here:

```
function deleteElementZero ( array )
{
 var i;
 var result;

 result = new Array ( Math.max(0, array.length-1) );
 i = 1;
 while ( i < array.length )
 {
  result [ i-1 ] = array [ i ];
  i = i + 1;
 }

 return result;
}
```

Exercise C

Create the function with the following header:

```
function deleteLastElement ( array )
```

The function is passed an array and returns a newly created array. The array that is returned by the function has all the elements of the parameter array with the exception of the last element of the parameter array. For example, the parameter array contained these elements: *5, 9, 3 4* then the array returned by the function would contain only these elements: 5, *9, 3*. So, the *length* of the array returned by the function is one less than the *length* of the parameter array. If the parameter array *length* is zero, then the function should return a newly created array whose *length* is also zero.

```
function deleteLastElement ( array )
{
 var i;
 var result;

 if ( array.length > 0 )
  result = new Array ( array.length - 1 );
  else
  result = new Array ( 0 );
 i = array.length-1;
 while ( i > 0 )
 {
  result [ i-1 ] = array [ i ];
  i = i - 1;
 }

 return result;
}
```

This function is similar to the previous function but, in this one, the last element of the array is removed rather than the first element. Again, the *if* statement could be replaced by the equivalent user of *Math.max* as shown in the previous example.

Exercise D

Create the function with the following header:

```
function locationInArray ( array, findMe )
```

The function is passed an array and a value. The function returns an integer. The return value is -1 if the value of *findMe* is not in the array. If the value of *findMe* is in the array, the function returns the element number of the array element whose value is equal to the value of *findMe*. If the value of *findMe* occurs more than one time in the array, the function should return the location of the match that is nearest to the beginning of the array.

```
function locationInArray ( array, findMe )
{
 var i;

 i = 0;
 while ( i < array.length && array[i] !== findMe ) { i = i + 1; }
 if ( i === array.length ) { i = -1; }
 return i;
}
```

If *findMe* occurs more than one time in the array, this function will return the component number of the first occurrence of *findMe* in the array. This happens because the loop has two stopping conditions:

- the entire array has been examined, or
- *findMe* has been found.

When either of these is *true*, the loop will stop. The *if* statement after the loop checks to see if *findMe* was located and, if not, sets the return value to -1. Can you figure out how to write this function so that it finds the last occurrence of *findMe* in the array?

Exercise E

Create the function with the following header:

```
function deleteKthElement ( array, k )
```

The function is passed an array and an integer *k*. The function returns an array. If the value of *k* is a valid subscript value for the parameter array (>=0 and <*array.length*), the function returns a new array that contains all the elements of the parameter array except element number *k*. That means that the array returned by the function will have one fewer elements than the parameter array. If the value of *k* is not a valid subscript value for the parameter array, the function returns a new array that contains all the elements of the parameter array.

```
function deleteKthElement ( array, k )
{
 var count;
 var i;
 var result;

 if ( k >=0 && k < array.length )
  result = new Array ( array.length - 1 );
  else
  result = new Array ( array.length );

 i     = 0;
 count = 0;
 while ( i < array.length )
 {
  if ( i !== k )
  {
   result [ count ] = array [ i ];
   count            = count + 1;
  }
  i = i + 1;
 }

 return result;
}
```

The function is passed an integer whose value is stored in *k*. The function will delete the element identified by the value but only if the value is a valid subscript for the array parameter. Recall that valid array subscripts have the same range as valid String character numbers: zero through *length-1*.

Exercise F

Create the function with the following header:

```
function deleteAllOccurrences ( array, findMe )
```

The function is passed an array and *findMe*, a value of any kind (*String*, number, etc.). The function returns an array. The array that is returned has all the values of the parameter array after all the *findMe* values have been deleted. If *findMe* is not in the parameter array, then the return value is a copy of the parameter array.

```
function deleteAllOccurrences ( array, findMe )
{
 var loc;
 var result;

 result = copy ( array );

 loc = locationInArray ( result, findMe );
 while ( loc >= 0 )
 {
  result = deleteKthElement ( result, loc );
  loc = locationInArray ( result, findMe );
 }
 return result;
}
```

This function uses the array copy function that you previously wrote. The function uses the *locationInArray* and *deleteKthElement* functions to simplify its implementation. If those helper functions were not used, this function would need to implement the *count- create-populate* strategy in order to delete all the array elements whose value is *findMe*.

Start of Today's Lecture

In this lecture we will review arrays and compare array use to *String* use. *Strings* and arrays are somewhat similar but have important differences. Older languages did not have a *String* data structure and text was stored as single characters in an array. If you ever look at the C language you will understand how this was done.

All modern languages directly support *Strings* and, in doing so, have moved away from the array storage of characters that was used in earlier languages. *Strings* are now different than arrays but still share some similarities.

Content

Strings can hold only characters. Arrays can hold any type of data: numbers, *Strings*, or as will be discussed in a future lecture, other arrays.

Length

Both arrays and *Strings* have a *length* property. The *length* property tells you the capacity – how many things can it store. If the variable *x* is a reference to an array, then *x.length* returns the number of elements in the array. An array can have zero or more elements.

If the variable *x* is a reference to a *String*, then *x.length* returns the number of characters in the *String*. There can be zero or more characters in a *String*.

Creation

A *String* is created by assigning a variable zero or more characters that are contained inside quote marks. For example,

```
hold = "abc":
```

Most languages only allow quote marks to be used. Javascript allows either quote marks or apostrophes. *Strings* can also be created using concatenation or calling one of the many built-in *String* functions such as *charAt, substring, toUpperCase,* etc.

In *Strings*, the capacity of the *String* and the contents of the *String* are both done in the same operation. The assignment statement shown above caused the system to allocate sufficient memory to hold three characters and then stored *abc* into that memory. This is different when using arrays. Creating an array has two steps:

- Create the capacity (how many elements will be in the array)
  Examples:

```
var x;
x = new Array ( 5 ); // create an array with a five element capacity
x = new Array ( 0 ); // create an array with a zero element capacity
```

- Assign content to each element
  Assigning content to an array is done by accessing each element of the array. Arrays can only be accessed one element at a time. This is frequently done using a loop but, for smaller arrays, it can be done through individual assignment statements.

  Examples:

```
var x;
x = new Array ( 3 ); // create an array with a three element capacity
x [ 0 ] =  5;
x [ 1 ] = -3;
x [ 2 ] = x [0]+ x [1];
```

- The Javascript language supports syntax that combines setting the capacity and assigning the content in a single statement. Most modern languages have this feature. This is useful when creating arrays that hold the names of the months, etc.
  Examples:
```
var x;
x = [ "desk", "chair" ]; // a two element array containing Strings
x = [];                  // an array with zero elements
```

Mutability

Neither *Strings* nor arrays can have their capacity changed. When a *String* is created or when an array is created, its capacity is fixed. Technically this is not true for Javascript arrays but you are required to conform to this restriction because the languages you will be using after this class will apply this rule. I don't want you to fall into bad habits.

Accessing

Accessing is retrieving a value and should be distinguished from *storing* which is changing a value.

Arrays can be accessed only one element at a time. The element that will be accessed is identified by the *subscript*: an integer whose value is greater than or equal to zero and less than the capacity (*length*) of the array. Accessing an element of an array uses this syntax:

```
arrayVariableName [ subscript ]
```

It is not possible to access more than one element of an array at a time. This is why most array processing is done inside loops. Each iteration of the loop processes one element of the array.

Strings can be accessed one or more characters at a time. The character to be accessed is identified by the *character number*: an integer whose value is greater than or equal to zero and less than the capacity (*length*) of the array. There are a couple of ways to access the characters in a *String*.

```
stringVariableName.charAt ( i )          // accesses one character
stringVariableName.substring ( i, j )    // copies characters i through (j-1)
stringVariableName.substring ( i, i+1 ) // copies ith character
```

Storing

Storing changes the value of a variable. *Strings* cannot be changed. This is referred to as being immutable. Once a *String* is created, neither its capacity nor its content can be modified. You must create a new *String* that contains the modification. You cannot modify an existing *String*. Neither *charAt* nor *substring* can appear to the left of the assignment operator (=).

Array elements can be modified. While the capacity of the array is not changeable, the values of any of the elements of the array can be modified at any time. Only one element can be modified in each assignment statement, though. Here is an example:

```
someArrayReference [ 2 ] = "Hello";
```

The effect of the assignment operator on an element of an array is the same as it is on any other variable. The value of the element is replaced by the value to the right of the assignment operator and the old value is gone forever.

Searching

Strings have a convenient, although oddly named, built-in function that searches the characters stored in the *String* named *indexOf*. This function does a case sensitive search and allows searching for a single character or a sequence of characters. In addition, you can control where

the search begins in the *String* by providing an additional character number parameter on the call to the function. If the search is successful, the value returned by the function is the character number where the match begins in the *String*. If the search is unsuccessful, the value -1 is returned by the function.

Searching an array must be done using a loop as shown above in Exercise D. Although Javascript does have an ability to do this, we will not be using it in this class. It is very important that you become comfortable with array processing and searching an array is an elementary array processing problem.

Problems

For each of the following *String* processing functions, write the equivalent array processing function. In the functions that you write, be careful not to modify the contents of the parameter array. Be sure to thoroughly test each function.

Problem 1

Here is a function that returns the count of the occurrences of *countMe* in the *String* source.

```
function countInString ( source, countMe )
{
 var i;
 var count;

 i = 0;
 count = 0;
 while ( i < source.length )
 {
  if ( source.charAt(i) === countMe ) { count = count + 1; }
  i = i + 1;
 }

 return count;
}
```

Write the equivalent function that has an array parameter and returns the number of elements in the array that have the value *countMe.*

```
function countInArray ( array, countMe )
{
 var i;
 var count;

// fill in the required statements here

return count;
}
```

Problem 2

Here is a function that is passed a *String*. It returns a *String* that contains all the characters of the parameter *String* but changes all *find* characters into *replace* characters. For example, if the function were called this way:

```
changeString("abcde","b","Z")
```

it would return *aZcdeZ*

```
function changeString ( source, find, replace )
{
 var ch;
 var i;
 var result;

 result = "";
 i = 0;
 while ( i < source.length )
 {
  ch = source.charAt ( i );
  if ( ch === find ) ch = replace;
  result = result + ch;
  i = i + 1;
 }

 return result;
}
```

Now write a function that is passed an array. It returns a newly created array that contains all the values of the parameter array but changes all *find* values into *replace* values.

```
function changeArray ( array, find, replace )
{
 var i;
 var result;

// fill in the required statements here

 return result;
}
```

Problem 3

Here is a function that is passed a *String*. It returns a *String* that contains all the characters of the parameter String but swaps character number *i* and character number *j*. For example, if the function were called this way:

```
swapCharacters("ABCDEF",3,2)
```

it would return *ABDCEF*

```
function swapCharacters ( source, i, j )
{
 var hold;
 var result;

 if ( i === j )
 {
  result = source;
 }
 else
 {
  hold = Math.min ( i, j );
  j    = Math.max ( i, j );
  i    = hold;

  result = source.substring ( 0, i )
         + source.charAt ( j )
         + source.substring ( i+1, j )
         + source.charAt ( i )
         + source.substring(j+1);
  }

 return result;
}
```

Now write a function that is passed an array. It returns a newly created array that contains all the values of the parameter array but with the values of elements *i* and *j* swapped. The values in the parameter array must not be changed in the function.

```
function swapElements ( array, i, j )
{
 var hold;
 var k;
 var result;

// fill in the required statements here

 return result;
}
```