Assignment Day 2 | 15th July 2020

Question 1:

Write a JS code which takes input from the user and logs it in the console.?

Ans: Javascript code which takes input from the user and logs it in the console.

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Get Text Input Field Value in JavaScript</title>

</head>

<body>

   <input type="text" placeholder="Type something..." id="myInput">

   <button type="button" onclick="getInputValue();">Get Value</button>


   <script>
```

```
        function getInputValue(){
            // Selecting the input element and get its value
            var inputVal =
document.getElementById("myInput").value;


            // Displaying the value
            alert(inputVal);
        }
    </script>
</body>
</html>
```

Question 2:

Explain with examples the remaining methods of String and Array

Ans:

STRINGS

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their length, to build and concatenate them using the + and += string operators, checking for the existence or location of substrings with the indexOf() method, or extracting substrings with the substring() method.

Creating strings

Strings can be created as primitives, from string literals, or as objects, using the String() constructor:

```
const string1 = "A string primitive";

const string2 = 'Also a string primitive';

const string3 = `Yet another string primitive`;

const string4 = new String("A String object");
```

String primitives and string objects can be used interchangeably in most situations. See "String primitives and String objects" below.

String literals can be specified using single or double quotes, which are treated identically, or using the backtick character `. This last form specifies a template literal: with this form you can interpolate expressions.

Character access

There are two ways to access an individual character in a string. The first is the charAt() method:

return 'cat'.charAt(1) // returns "a"

The other way (introduced in ECMAScript 5) is to treat the string as an array-like object, where individual characters correspond to a numerical index:

return 'cat'[1] // returns "a"

When using bracket notation for character access, attempting to delete or assign a value to these properties will not succeed. The properties involved are neither writable nor configurable. (See Object.defineProperty() for more information.)

Comparing strings

In C, the strcmp() function is used for comparing strings. In JavaScript, you just use the less-than and greater-than operators:

```
let a = 'a'
let b = 'b'
if (a < b) { // true
  console.log(a + ' is less than ' + b)
} else if (a > b) {
  console.log(a + ' is greater than ' + b)
```

```
} else {
  console.log(a + ' and ' + b + ' are equal.')
}
```

A similar result can be achieved using the localeCompare() method inherited by String instances.

Note that a == b compares the strings in a and b for being equal in the usual case-sensitive way. If you wish to compare without regard to upper or lower case characters, use a function similar to this:

```
function isEqual(str1, str2)
{
   return str1.toUpperCase() === str2.toUpperCase()
} // isEqual
```

Upper case is used instead of lower case in this function, due to problems with certain UTF-8 character conversions.

String primitives and String objects

Note that JavaScript distinguishes between String objects and primitive string values. (The same is true of Boolean and Numbers.)

String literals (denoted by double or single quotes) and strings returned from String calls in a non-constructor context (that is, called without using the new keyword) are primitive strings. JavaScript automatically converts primitives to String objects, so that it's possible to use String object methods for primitive strings. In contexts where a method is to be invoked on a primitive string or a property lookup occurs, JavaScript will automatically wrap the string primitive and call the method or perform the property lookup.

```
let s_prim = 'foo'

let s_obj = new String(s_prim)


console.log(typeof s_prim) // Logs "string"

console.log(typeof s_obj)  // Logs "object"
```

String primitives and String objects also give different results when using eval(). Primitives passed to eval are treated as source code; String objects are treated as all other objects are, by returning the object. For example:

```
let s1 = '2 + 2'             // creates a string primitive

let s2 = new String('2 + 2')  // creates a String object
```

```
console.log(eval(s1))        // returns the number 4

console.log(eval(s2))        // returns the string "2 + 2"
```

For these reasons, the code may break when it encounters String objects when it expects a primitive string instead, although generally, authors need not worry about the distinction.

A String object can always be converted to its primitive counterpart with the valueOf() method.

```
console.log(eval(s2.valueOf()))  // returns the number 4
```

Escape notation

Special characters can be encoded using escape notation:

Code	Output
\XXX	(where XXX is 1–3 octal digits; range of 0–377) ISO-8859-1 character / Unicode code point between U+0000 and U+00FF
\'	single quote
\"	double quote
\\	backslash
\n	new line

\r carriage return

\v vertical tab

\t tab

\b backspace

\f form feed

\uXXXX (where XXXX is 4 hex digits; range of 0x0000–0xFFFF)
UTF-16 code unit / Unicode code point between U+0000 and U+FFFF

\u{X} ... \u{XXXXXX}

(where X…XXXXXX is 1–6 hex digits; range of 0x0–0x10FFFF)
UTF-32 code unit / Unicode code point between U+0000 and U+10FFFF

\xXX

(where XX is 2 hex digits; range of 0x00–0xFF) ISO-8859-1 character / Unicode code point between U+0000 and U+00FFLong literal strings

Sometimes, your code will include strings which are very long. Rather than having lines that go on endlessly, or wrap at the whim of your editor, you may wish to specifically break the string into multiple lines in the source code without affecting the actual string contents. There are two ways you can do this.

Method 1

You can use the + operator to append multiple strings together, like this:

```
let longString = "This is a very long string which needs " +
                 "to wrap across multiple lines because " +
                 "otherwise my code is unreadable."
```

Method 2

You can use the backslash character (\) at the end of each line to indicate that the string will continue on the next line. Make sure there is no space or any other character after the backslash (except for a line break), or as an indent; otherwise it will not work.

That form looks like this:

```
let longString = "This is a very long string which needs \
to wrap across multiple lines because \
otherwise my code is unreadable."
```

Both of the above methods result in identical strings.

Constructor

String()

Creates a new String object. It performs type conversion when called as a function, rather than as a constructor, which is usually more useful.

Static methods

String.fromCharCode(num1 [, ...[, numN]])

Returns a string created by using the specified sequence of Unicode values.

String.fromCodePoint(num1 [, ...[, numN)

Returns a string created by using the specified sequence of code points.

String.raw()

Returns a string created from a raw template string.

Instance properties

String.prototype.length

Reflects the length of the string. Read-only.

Instance methods

String.prototype.charAt(index)

Returns the character (exactly one UTF-16 code unit) at the specified index.

String.prototype.charCodeAt(index)

Returns a number that is the UTF-16 code unit value at the given index.

String.prototype.codePointAt(pos)

Returns a nonnegative integer Number that is the code point value of the UTF-16 encoded code point starting at the specified pos.

String.prototype.concat(str [, ...strN])

Combines the text of two (or more) strings and returns a new string.

String.prototype.includes(searchString [, position])

Determines whether the calling string contains searchString.

String.prototype.endsWith(searchString [, length])

Determines whether a string ends with the characters of the string searchString.

String.prototype.indexOf(searchValue [, fromIndex])

Returns the index within the calling String object of the first occurrence of searchValue, or -1 if not found.

String.prototype.lastIndexOf(searchValue [, fromIndex])

Returns the index within the calling String object of the last occurrence of searchValue, or -1 if not found.

String.prototype.localeCompare(compareString [, locales [, options]])

Returns a number indicating whether the reference string compareString comes before, after, or is equivalent to the given string in sort order.

String.prototype.match(regexp)

Used to match regular expression regexp against a string.

String.prototype.matchAll(regexp)

Returns an iterator of all regexp's matches.

String.prototype.normalize([form])

Returns the Unicode Normalization Form of the calling string value.

String.prototype.padEnd(targetLength [, padString])

Pads the current string from the end with a given string and returns a new string of the length targetLength.

String.prototype.padStart(targetLength [, padString])

Pads the current string from the start with a given string and returns a new string of the length targetLength.

String.prototype.repeat(count)

Returns a string consisting of the elements of the object repeated count times.

String.prototype.replace(searchFor, replaceWith)

Used to replace occurrences of searchFor using replaceWith. searchFor may be a string or Regular Expression, and replaceWith may be a string or function.

String.prototype.replaceAll(searchFor, replaceWith)

Used to replace all occurrences of searchFor using

replaceWith. searchFor may be a string or Regular Expression, and replaceWith may be a string or function.

String.prototype.search(regexp)

Search for a match between a regular expression regexp and the calling string.

String.prototype.slice(beginIndex[, endIndex])

Extracts a section of a string and returns a new string.

String.prototype.split([sep [, limit]])

Returns an array of strings populated by splitting the calling string at occurences of the substring sep.

String.prototype.startsWith(searchString [, length])

Determines whether the calling string begins with the characters of string searchString.

String.prototype.substr()

Returns the characters in a string beginning at the specified location through the specified number of characters.

String.prototype.substring(indexStart [, indexEnd])

Returns a new string containing characters of the calling string from (or between) the specified index (or indeces).

String.prototype.toLocaleLowerCase([locale, ...locales])

The characters within a string are converted to lowercase while respecting the current locale.

For most languages, this will return the same as toLowerCase().

String.prototype.toLocaleUpperCase([locale, ...locales])

The characters within a string are converted to uppercase while respecting the current locale.

For most languages, this will return the same as toUpperCase().

String.prototype.toLowerCase()

Returns the calling string value converted to lowercase.

String.prototype.toString()

Returns a string representing the specified object. Overrides the Object.prototype.toString() method.

String.prototype.toUpperCase()

Returns the calling string value converted to uppercase.

String.prototype.trim()

Trims whitespace from the beginning and end of the string. Part of the ECMAScript 5 standard.

String.prototype.trimStart()

Trims whitespace from the beginning of the string.

String.prototype.trimEnd()

Trims whitespace from the end of the string.

String.prototype.valueOf()

Returns the primitive value of the specified object. Overrides the Object.prototype.valueOf() method.

String.prototype.@@iterator()

Returns a new Iterator object that iterates over the code points of a String value, returning each code point as a String value.

ARRAYS

Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's length can change at any time, and data can be stored at non-contiguous locations in the array, JavaScript arrays are not guaranteed to be dense; this depends on how the programmer chooses to use them. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays.

Arrays cannot use strings as element indexes (as in an

associative array) but must use integers. Setting or accessing via non-integers using bracket notation (or dot notation) will not set or retrieve an element from the array list itself, but will set or access a variable associated with that array's object property collection. The array's object properties and list of array elements are separate, and the array's traversal and mutation operations cannot be applied to these named properties.

Common operations

Create an Array

```
let fruits = ['Apple', 'Banana']


console.log(fruits.length)
// 2
```

Access an Array item using the index position

```
let first = fruits[0]
// Apple


let last = fruits[fruits.length - 1]
```

// Banana

Loop over an Array

```
fruits.forEach(function(item, index, array) {
  console.log(item, index)
})
```

// Apple 0

// Banana 1

Add an item to the end of an Array

```
let newLength = fruits.push('Orange')
```

// ["Apple", "Banana", "Orange"]

Remove an item from the end of an Array

```
let last = fruits.pop() // remove Orange (from the end)
```

// ["Apple", "Banana"]

Remove an item from the beginning of an Array

```
let first = fruits.shift() // remove Apple from the front
```

// ["Banana"]

Add an item to the beginning of an Array

let newLength = fruits.unshift('Strawberry') // add to the front

// ["Strawberry", "Banana"]

Find the index of an item in the Array

fruits.push('Mango')

// ["Strawberry", "Banana", "Mango"]

let pos = fruits.indexOf('Banana')

// 1

Remove an item by index position

let removedItem = fruits.splice(pos, 1) // this is how to remove an item

// ["Strawberry", "Mango"]

Remove items from an index position

let vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot']

console.log(vegetables)

// ["Cabbage", "Turnip", "Radish", "Carrot"]

let pos = 1

let n = 2

let removedItems = vegetables.splice(pos, n)

// this is how to remove items, n defines the number of items to be removed,

// starting at the index position specified by pos and progressing toward the end of array.

console.log(vegetables)

// ["Cabbage", "Carrot"] (the original array is changed)

console.log(removedItems)

// ["Turnip", "Radish"]

Copy an Array

let shallowCopy = fruits.slice() // this is how to make a copy

// ["Strawberry", "Mango"]

Accessing array elements

JavaScript arrays are zero-indexed: the first element of an array is at index 0, and the last element is at the index equal to the value of the array's length property minus 1.

Using an invalid index number returns undefined.

```
let arr = ['this is the first element', 'this is the second element', 'this is the last element']

console.log(arr[0])              // logs 'this is the first element'

console.log(arr[1])              // logs 'this is the second element'

console.log(arr[arr.length - 1]) // logs 'this is the last element'
```

Array elements are object properties in the same way that toString is a property (to be specific, however, toString() is a method). Nevertheless, trying to access an element of an array as follows throws a syntax error because the property name is not valid:

```
console.log(arr.0) // a syntax error
```

There is nothing special about JavaScript arrays and the properties that cause this. JavaScript properties that begin with a digit cannot be referenced with dot notation and must be accessed using bracket notation.

For example, if you had an object with a property named '3d', it can only be referenced using bracket notation.

```
let years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]

console.log(years.0)   // a syntax error

console.log(years[0])  // works properly

renderer.3d.setTexture(model, 'character.png')     // a syntax error

renderer['3d'].setTexture(model, 'character.png')  // works properly
```

Note that in the 3d example, '3d' had to be quoted. It's possible to quote the JavaScript array indexes as well (e.g., years['2'] instead of years[2]), although it's not necessary.

The 2 in years[2] is coerced into a string by the JavaScript engine through an implicit toString conversion. As a result, '2' and '02' would refer to two different slots on the years object, and the following example could be true:

```
console.log(years['2'] != years['02'])
```

Relationship between length and numerical properties

A JavaScript array's length property and numerical properties are connected.

Several of the built-in array methods (e.g., join(), slice(), indexOf(), etc.) take into account the value of an array's length property when they're called.

Other methods (e.g., push(), splice(), etc.) also result in updates to an array's length property.

```
const fruits = []

fruits.push('banana', 'apple', 'peach')


console.log(fruits.length) // 3
```

When setting a property on a JavaScript array when the property is a valid array index and that index is outside the current bounds of the array, the engine will update the array's length property accordingly:

```
fruits[5] = 'mango'

console.log(fruits[5])           // 'mango'

console.log(Object.keys(fruits))  // ['0', '1', '2', '5']

console.log(fruits.length)       // 6
```

Increasing the length.

```
fruits.length = 10

console.log(fruits)              // ['banana', 'apple', 'peach',
undefined, 'mango', <5 empty items>]

console.log(Object.keys(fruits)) // ['0', '1', '2', '5']

console.log(fruits.length)       // 10

console.log(fruits[8])           // undefined
```

Decreasing the length property does, however, delete elements.

```
fruits.length = 2

console.log(Object.keys(fruits)) // ['0', '1']

console.log(fruits.length)       // 2
```

This is explained further on the Array.length page.

Creating an array using the result of a match

The result of a match between a RegExp and a string can create a JavaScript array. This array has properties and elements which provide information about the match. Such an array is returned by RegExp.exec(), String.match(), and String.replace().

To help explain these properties and elements, see this

example and then refer to the table below:

// Match one d followed by one or more b's followed by one d

// Remember matched b's and the following d

// Ignore case

const myRe = /d(b+)(d)/i

const myArray = myRe.exec('cdbBdbsbz')

The properties and elements returned from this match are as follows:

Property/Element Description Example

input

Read only The original string against which the regular expression was matched. "cdbBdbsbz"

index

Read only The zero-based index of the match in the string. 1

[0]

Read only The last matched characters. "dbBd"

[1], ...[n]

Read only Elements that specify the parenthesized substring

matches (if included) in the regular expression. The number of possible parenthesized substrings is unlimited. [1]: "bB"

[2]: "d"

Constructor

Array()

Creates a new Array object.

Static properties

get Array[@@species]

The constructor function is used to create derived objects.

Static methods

Array.from()

Creates a new Array instance from arrayLike, an array-like or iterable object.

Array.isArray()

Returns true if value is an array, or false otherwise.

Array.of()

Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

Instance properties

Array.prototype.length

Reflects the number of elements in an array.

Array.prototype[@@unscopables]

A symbol containing property names to exclude from a with binding scope.

Instance methods

Array.prototype.concat()

Returns a new array that is this array joined with other array(s) and/or value(s).

Array.prototype.copyWithin()

Copies a sequence of array elements within the array.

Array.prototype.entries()

Returns a new Array Iterator object that contains the key/value pairs for each index in the array.

Array.prototype.every()

Returns true if every element in this array satisfies the testing callbackFn.

Array.prototype.fill()

Fills all the elements of an array from a start index to an end index with a static value.

Array.prototype.filter()

Returns a new array containing all elements of the calling array for which the provided filtering callbackFn returns true.

Array.prototype.find()

Returns the found element in the array, if some element in the array satisfies the testing callbackFn, or undefined if not found.

Array.prototype.findIndex()

Returns the found index in the array, if an element in the array satisfies the testing callbackFn, or -1 if not found.

Array.prototype.forEach()

Calls a callbackFn for each element in the array.

Array.prototype.includes()

Determines whether the array contains valueToFind, returning true or false as appropriate.

Array.prototype.indexOf()

Returns the first (least) index of an element within the array equal to searchElement, or -1 if none is found.

Array.prototype.join()

Joins all elements of an array into a string.

Array.prototype.keys()

Returns a new Array Iterator that contains the keys for each index in the array.

Array.prototype.lastIndexOf()

Returns the last (greatest) index of an element within the array equal to searchElement, or -1 if none is found.

Array.prototype.map()

Returns a new array containing the results of calling callbackFn on every element in this array.

Array.prototype.pop()

Removes the last element from an array and returns that element.

Array.prototype.push()

Adds one or more elements to the end of an array, and returns the new length of the array.

Array.prototype.reduce()

Apply a callbackFn against an accumulator and each value of the array (from left-to-right) as to reduce it to a single value.

Array.prototype.reduceRight()

Apply a callbackFn against an accumulator and each value of the array (from right-to-left) as to reduce it to a single value.

Array.prototype.reverse()

Reverses the order of the elements of an array in place. (First becomes the last, last becomes first.)These methods modify the array:

Array.prototype.shift()

Removes the first element from an array and returns that element.

Array.prototype.slice()

Extracts a section of the calling array and returns a new array.

Array.prototype.some()

Returns true if at least one element in this array satisfies the provided testing callbackFn.

Array.prototype.sort()

Sorts the elements of an array in place and returns the array.

Array.prototype.splice()

Adds and/or removes elements from an array.

Array.prototype.toLocaleString()

Returns a localized string representing the array and its elements. Overrides the Object.prototype.toLocaleString() method.

Array.prototype.toString()

Returns a string representing the array and its elements. Overrides the Object.prototype.toString() method.

Array.prototype.unshift()

Adds one or more elements to the front of an array, and returns the new length of the array.

Array.prototype.values()

Returns a new Array Iterator object that contains the values for each index in the array.

Array.prototype[@@iterator]()

Returns a new Array Iterator object that contains the values for

each index in the array.

Examples

Creating an array

The following example creates an array, msgArray, with a length of 0, then assigns values to msgArray[0] and msgArray[99], changing the length of the array to 100.

```
let msgArray = []

msgArray[0] = 'Hello'

msgArray[99] = 'world'


if (msgArray.length === 100) {

  console.log('The length is 100.')

}
```

Creating a two-dimensional array

The following creates a chessboard as a two-dimensional array of strings. The first move is made by copying the 'p' in board[6][4] to board[4][4]. The old position at [6][4] is made blank.

```
let board = [

  ['R','N','B','Q','K','B','N','R'],
```

```
  ['P','P','P','P','P','P','P','P'],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  ['p','p','p','p','p','p','p','p'],
  ['r','n','b','q','k','b','n','r'] ]

console.log(board.join('\n') + '\n\n')

// Move King's Pawn forward 2
board[4][4] = board[6][4]
board[6][4] = ' '
console.log(board.join('\n'))
```

Here is the output:

```
R,N,B,Q,K,B,N,R
P,P,P,P,P,P,P,P
 , , , , , , ,
 , , , , , , ,
```

, , , , , , ,

, , , , , , ,

p,p,p,p,p,p,p,p

r,n,b,q,k,b,n,r

R,N,B,Q,K,B,N,R

P,P,P,P,P,P,P,P

, , , , , , ,

, , , , , , ,

, , , ,p, , ,

, , , , , , ,

p,p,p,p, ,p,p,p

r,n,b,q,k,b,n,r

Using an array to tabulate a set of values

```
values = []
for (let x = 0; x < 10; x++){
 values.push([
  2 ** x,
  2 * x ** 2
 ])
```

}

console.table(values)

Results in

0	1	0
1	2	2
2	4	8
3	8	18
4	16	32
5	32	50
6	64	72
7	128	98
8	256	128
9	512	162

(The first column is the (index))