# CSCE 120: Learning To Code

## Module 10.0: Processing Data II
## Array & String Processing

This module revisits arrays and strings and introduces you to several of the standard functions associated with them that can be used to process string and array data.

## Introduction

The JavaScript language provides many useful functions for processing data stored in an array and in strings. We've already seen how we can manually process data by writing our own loops and processing code. However, its generally preferable to utilize the library functions that already exist. It makes our code simpler and more maintainable. Using standard libraries also makes our code more robust and usually more efficient. The standard library functions have been well-tested and optimized. We could never hope to repeat the work and effort that has gone in to developing them.

In this module, we'll take a look at several of the most useful functions for processing data in arrays and strings.

## Array Processing

JavaScript defines several other useful ways to interact with data stored in arrays. Documentation on each can be found here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

### Modifying Arrays

There are several ways that data can be added and/or removed from arrays in JavaScript. The `splice()` function allows you to add and remove elements at any index of the array,

`arr`. This function takes a variable number of arguments, but the first two are required.

```
arr.splice(index, numRemove, [item1, item2, ..., itemN])
```

- `index` specifies *where* you want the add/remove operation to occur.
- `numRemove` specifies how many items you want to remove. If this number is zero, it will not remove any items and instead only add items.
- After the first two arguments, you can provide a variable number of elements to be inserted starting at `index`. Each item is delimited by a comma. The notation using the square brackets above is a convention used to indicate that these arguments are optional.

To understand this better, consider the following examples.

```
var arr = [10, 20, 30, 40];

//add 5, 15, 25 to the beginning of the array,
//do not remove anything
arr.splice(0, 0, 5, 15, 25); //[ 5, 15, 25, 10, 20, 30, 40 ]

//insert 5 at the end
arr.splice(arr.length, 5); //[10, 20, 30, 40, 5]

//add 4, 3, 5 to the end of the array:
arr.splice(arr.length, 0, 4, 3, 5); //[10, 20, 30, 40, 4, 3, 5 ]

//insert 5 in the middle:
arr.splice(2, 0, 5); //[10, 20, 5, 30, 40]

//remove the first three elements:
arr.splice(0, 3);  //[40]

//remove the last two elements
arr.splice(arr.length-2, 2); //[10, 20]

//alternatively, we can provide negative indices to indicate a
//position relative to the end of the array:
//remove the last two elements
arr.splice(-2, 2); //[10, 20]

//remove the first two elements, insert 42
arr.splice(0, 2, 42); // [42, 30, 40]

//remove the first two elements, insert 42, 43;
//essentially *replace* elements:
arr.splice(0, 2, 42, 43); // [42, 43, 30, 40]
```

The `splice` function is the most general purpose add/remove function for arrays. It allows you to add/remove any number of elements to any part of the array. There are several other methods that allow you to add or remove elements from arrays with more specific behavior.

Two examples of such functions are `push` and `pop`. The `push` function adds a single element to the *end* of an array while the `pop` function removes (and returns) the element at the end of the array. Some examples:

```
var arr = [10, 20];
arr.push(30); //[10, 20, 30];
var x;
x = arr.pop(); //x holds 30, arr is [10, 20]
x = arr.pop(); //x holds 20, arr is [10]
x = arr.pop(); //x holds 10, arr is [] (empty)

//popping from an empty array results in undefined:
x = arr.pop(); //x is undefined

//you can push multiple elements at once:
arr.push(5); //[5]
arr.push(15, 25); //[5, 15, 25]
```

Using `push` and `pop` allows you to treat an array like a *stack* data structure. A stack is a data structure that allows you to place elements into a collection in a Last-In First-Out (LIFO) order and is a fundamental data structure in computer science.

You can also use `unshift` and `shift` which work like `push`/`pop` but they operate on the *beginning* of the array. The `unshift` function adds an element (or elements) to the beginning of the array while `shift` removes (and returns) an element from the beginning of the array.

```
var arr = [10, 20, 30];

arr.unshift(5); //[5, 10, 20, 30]
arr.unshift(15); //[15, 5, 10, 20, 30]

x = arr.shift(); //x holds 15, arr is [5, 10, 20, 30]
x = arr.shift(); //x holds 5, arr is [10, 20, 30]

//you can unshift multiple elements:
arr.unshift(100, 200);

//shifting from an empty array results in undefined:
x = [].shift; //x is undefined
```

## Map

We have previously examined the `forEach` array function and the jQuery `each` function that allow you to apply a callback to each element in an array. JavaScript provides a similar useful function called `map`. The `map` function allows you to apply a callback to each element in the array, but instead of modifying the contents of the array, the result of the `map` function is a *new* array. The callback is expected to process each element in the array and return a new result (that is then stored in the new array). The contents of

the original array are unmodified.

```
function addOne(value, index, array) {
  return value + 1;
}

var arr = [2, 4, 9, 10];
var newArr = arr.map(addOne);

//or, preferably, as an anonymous function:
var newArr = arr.map(function(value, index, array) {
  return value + 1;
});

//arr is still [2, 4, 9, 10]
//newArr is now [3, 5, 10, 11]
```

The callback passed to the `map` takes three arguments: the `value` of the element, its `index` and the `array` on which the `map` function was applied. In the example, we only use the value.

## Reduce

The `reduce` function allows you to aggregate array values into one final result. It takes two arguments: a callback to apply to each element as well as an (optional) initial value. The callback in this case is expected to have 4 arguments:

`function(previousValue, currentValue, index, array) {...}`

- The `previousValue` is the value carried over from the last call to the callback
- The `currentValue` is the value of the element in the array currently being processed
- `index` is the index of the `currentValue`
- `array` is the original array on which the `reduce` function was invoked on

The `reduce` function works like this: it iteratively applies the callback to each element in the array working left-to-right starting at the first element in the array. The `previousValue` is carried over from call to call allowing you to aggregate the data, for example summing values in an array:

```
var arr = [2, 4, 9, 10];

var sum = arr.reduce(function(prevValue, currValue, index, array) {
  return prevValue + currValue;
}, 0); //initial value: 0

//sum is now 25
```

This is where the second (optional) argument comes into play with the `reduce` function. If we want to compute a sum, then our initial value should be zero. If no initial value is provided, the `reduce` function starts at the second element with the initial value taking on the value of the first element.

### Filter

The `filter` function allows you to process an array and retain or remove elements based on any property/logic. The `filter` function takes a callback that processes each element and is expected to return `true` if the element is to be retained or `false` if it should not be. The `filter` function results in a new array (the original array is unchanged).

```
var arr = [2, 4, 9, 10, 51, 92, 7, 3];

var newArr = arr.filter(function(value, index, array) {
  if(value % 2 == 0) {
    //value is even, return true to keep it
    return true;
  } else {
    return false;
  }
});

arr; // still [2, 4, 9, 10, 51, 92, 7, 3]
newArr; //[ 2, 4, 10, 92 ]
```

## String Processing

String data can also be manipulated using standard JavaScript functions. Here we only highlight a few of the most useful functions. Full documentation on each as well as a more complete list of string functions can be found here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

For each of the following, assume that `s` is a string.

- `s.charAt(i)` – returns a single character at the index $i$ (strings are indexed start-

ing from 0 just like arrays).

- `s.search(a)` – searches for the string $a$ within $s$, returns the index at which it is found, $-1$ if it is not.
- `s.substr(start, length)` – returns a *substring* of $s$ from the given `start` index with the given `length`.
- `s.substring(start, end)` – returns a substring of $s$ from the given `start` index (indcluive) to the given `end` index (exclusive)
- `s.split(delimiter)` – returns an *array* of strings which are *tokens* of $s$ split by the given `delimiter`

```
var str = "Hello World!";
var x;

x = str.charAt(0); //"H"
x = str.charAt(4); //"o"
x = str.charAt(str.length-1); //"!"

x = str.search("H"); //0
x = str.search("W"); //6
x = str.search("X"); //-1, no instance found
//search is case sensitive
x = str.search("w"); //-1

x = str.substr(0, 5); //start at 0, length=5, "Hello"
x = str.substr(6, 5); //"World"
x = str.substr(6); //"World!"

x = str.substring(0, 2); //"He"
x = str.substring(6, 9); //"Wor"
x = str.substring(11); //"!"

str = "Smith,John,555-867-5309,jsmith@unl.edu";
var tokens = str.split(",");
//tokens is [ 'Smith', 'John', '555-867-5309', 'jsmith@unl.edu' ]
```

## Polyfills & Shims

Technology is always changing and progressing as are the tools, libraries and languages that its built on. Programming languages are constantly having new features and library functions added in each new version.

ECMAScript is the official standard that defines the JavaScript language. Most recently,

version 6 was finalized and standardized in June 2015. Though it added many new features, using these features in our applications can be problematic. Certain features may not be supported in all web browsers. Even if they are, there is no guarantee that all users have upgraded to the latest browser versions. It may not even be possible for some users to upgrade browsers if they are on old Operating Systems or old versions are necessary for legacy software.

In general, we cannot assume that all users will have browsers that support the features we wish to use in our application. There are, however, workarounds that can be used so that we can use new or future features and functionality even when using an older version of ECMAScript that does not support it.

One way of doing this is to use a *polyfill*. A polyfill is a piece of code that implements or extends a feature that is *expected* to be available in the browser or expected to be available in a future version. The way this generally works is that we can include code to check to see if a certain feature is supported or available (or check a browser version if necessary). If it is not, then we can initialize code that implements that feature. If the feature is available, the code is not initialized and the "native" functionality is used. In this way you can "future-proof" your application without having to worry about browser support.

As an example, in ECMAScript 5.1 and earlier, the only function provided to search an array of elements is to use the `indexOf(item)` function which returns the index of the first element that equals `item`. If the `item` does not exist, the function returns $-1$. However, this function is extremely limited. It can only be used to search arrays of strings or numbers. Ideally, we would have a general search function that took a callback so as to search for any type of element using any logic that we want.

ECMAScript 6 has added such a function, `find` (the documentation is available here: http://www.2ality.com/2014/05/es6-array-methods.html). If we wanted to take advantage of this function to search arrays, we would need to be assured that the browser supported ECMAScript 6.[1] A polyfill would look something like the following.

[1] As of version 0.7.2, Light Table only supports ECMAScript 5.1, so we can't even use it in our IDE yet.

```
if (!Array.prototype.find) {
  Array.prototype.find = function(predicate) {
    if (this == null) {
      throw new TypeError('Array.prototype.find called on null or undefined');
    }
    if (typeof predicate !== 'function') {
      throw new TypeError('predicate must be a function');
    }
    var list = Object(this);
    var length = list.length >>> 0;
    var thisArg = arguments[1];
    var value;

    for (var i = 0; i < length; i++) {
      value = list[i];
      if (predicate.call(thisArg, value, i, list)) {
        return value;
      }
    }
    return undefined;
  };
```

The first line checks to see if `Array` has a `find` function. If it does, the rest of the code has no effect. If it doesn't, then the rest of the code *extends* the functionality of `Array` to include a `find` function. A full list of functions supported by each version of ECMAScript (including future versions under consideration) can be found here: http://kangax.github.io/compat-table/es5/.

A related concept is a *shim*. A shim is like a polyfill in that it adds or extends functionality to JavaScript, but the difference is that a polyfill adds functionality that is expected to (eventually) be supported while a shim adds functionality that is not standard or expected to eventually be standard. Shims are often used to change or augment standard API behavior rather than introduce new functionality.