Collections Cheatsheet

Collections - Arrays

What is an Array?

- An array is an ordered list of values; these values can be strings, booleans, numbers... even other arrays.
- The values within an array, called **elements**, are accessed by their position (via a value called an**index**) within the array.
- An array can be defined by enclosing a list of values within square braces, like so : `var myArray = ['a','b','c','d']`
- To retrieve the value at some index `i` from an array, add `[`*i*`]` to the end of the array. e.g.`myArray[2]`
- To edit the value at some index `i`, simply act as if you were assigning a variable. e.g. `myArray[1] = 'f'`

Adding Complexity - Nested Arrays

- As mentioned above, arrays can contain other arrays as elements. This process of putting arrays inside other arrays (or just generally, putting things inside other things) is called **nesting**.
- To retrieve a value from a nested array, use one set of square braces for every level of nesting. The first set should hold the element's index in the outermost array, the second set should hold the index in the next-outermost, etc. If you were working with the following nested array,

```
  var myNestedArray = [['a','b','c'],                         ['d','e','f'],
['g','h','i']];
```

  you could access the element 'f' by writing `myNestedArray[1][2]`.

- Editing a value in a nested array is exactly like editing a value in a non-nested array; the only difference is how you reference the value that you want to change. e.g. `myNestedArray[0][3] = 'z';`

Additional Array Features

- In addition to storing a set of values, arrays also have a number of in-built properties and functions that they can use.
- `.length` gives you the length of the array you call it on.
- `.push()` adds a new element to the end of an array, and returns that element.
- `.pop()` removes the last element in an array, and returns that element.
- `.indexOf()` searches within your array for the first element that matches its parameter, and returns the index of that match; if no match is found, it returns -1.

Iterating Over Arrays

Iterating with Loops

- `for` loops are an easy way to iterate through an array. The following will execute an arbitrary function `someFunction` for every element in array `myArray`, from left to right.

```
  for (var i = 0; i < myArray.length; i += 1) {       someFunction(myArray[i]);
}
```

- To change the way that you iterate through the array, just change the settings of your `for` loop.

Iterator Functions

- `.map()` creates a new array with the results of calling a provided function on every element in this array.
- `.forEach()` executes some function once for each element in the array it's called on.

Collections - Associative Arrays

Drawbacks of Ordinary Arrays

- A typical array works by referenceing elements solely based on their position, e.g. "the first element, the second element ..." etc. But if the elements are ever rearranged, all of the references to specific elements need to be update.

- An associative array generates an enduring relationship between a reference (called a **key**) and the value that it refers to. Each key-value pairing is totally independent of every other pairing.

Associative Arrays in JavaScript

- An associative array can be defined by enclosing a list of key-value pairs in curly braces (`{...}`). Each key-value pair is written as `someKey : someValue`, and each pair is separated by commas.
- To retrieve the value that's tied to a particular key, add `[`*key*`]` to the end of the associative array. e.g. `myassociativeArray['myKey']`
- To edit the value that's tied to a particular key, assign a value just like you would for an ordinary array. e.g. `myAssociativeArray['myKey'] = 'aValue'`
- Adding a new key-value pair to an associative array is easy - it looks just like an assignment operation. e.g. `myAssociativeArray['someNewKey'] = 'someNewValue'`
- Nesting for associative arrays works *exactly* like it does for ordinary arrays.