JavaScript

Arrays

An array is a collection object that has a sequence of items you can access and modify. The array is assigned to a variable, and you can access its items by using its indexer, which is square brackets ([ ]). Because the collection of items is in one variable, you can easily pass the array to a function. You can also loop through the items in the array as needed.

Creating and populating an array

There are three ways to create an array. It doesn't matter which method you choose, although your choice typically will be based on the array implementation in the program:

- **Inserting items with the indexer** The array is created by using the *new* keyword, which creates an instance of the Array object. After the array is created and assigned to a variable, items are added to the array by using the index number, which is zero based. For inserting new items, the index number must be the size of the array.

  For example, after the array is created, its size is zero, so zero is used to insert the first item. Using this method, items can be added anywhere in the program. Note that if you use an index number that is higher than the quantity of items that currently exist, you add empty items to the array. For example, if you currently have only one item in the array but specify an index number of 2, you will add your item with an index number of 2, and an empty item will be added at index number 1. The following is an example of creating the array and adding items:

```
var pizzaParts = new Array();
pizzaParts[0] = 'pepperoni';
pizzaParts[1] = 'onion';
pizzaParts[2] = 'bacon';
```

- **Condensed array** The array is created by using the *new* keyword, which creates an instance of the Array object, and all items are passed into the Array object's constructor. The condensed method is convenient, but you need to know all items at the time you create the array. The following is an example of creating the populated array:

```
var pizzaParts = new Array('pepperoni', 'onion', 'bacon');
```

- **Literal array** The array is created by supplying the item list, enclosed in square brackets. This is very similar to the condensed array; it just requires less typing. The following is an example of the literal array:

```
var pizzaParts = ['pepperoni', 'onion', 'bacon'];
```

Accessing the array items

To access the items in the array, use the indexer. Remember that the array is zero-based, and if you try using a number that's greater than the quantity of items in the array, a value of undefined is returned. The following example retrieves the onion:

```
var secondItem = pizzaParts[1];
```

Modifying the array items

You also use the indexer when you want to modify the items in the array. If you try using a number that's greater than the quantity of items in the array, no exception is thrown. Instead, the item is added to the array, and the array size grows to the number you used plus one. The following example modifies the onion by setting its value to cheese:

```
pizzaParts[1] = 'cheese';
```

Remark: Consistently to the fact that variable in JavaScript does not have declared type, array can be **heterogeneous**. For instance,

```
var arr = [5, "John", function (a, b) { return a + b }];
```

So, for instance, we can call `arr[2](3, 5)` to produce 8.

Understanding array properties

Each piece of data objects can hold is called a *property*. Some properties are read-only, whereas others are readable and writeable. The Array object has one property that you'll use often, the *length* property. This property is read-only and returns the quantity of items in the array. For example, an array with two items returns 2. The length property is useful when looping through the items in the array. The following example code demonstrates the length property:

```
for(var i=0; i < pizzaParts.length; i++){ alert(pizzaParts[i]); }
```

Using array methods

Objects can have their own functions; each object function is called a *method*. The Array object has the following useful methods:

- **concat** Joins two or more arrays and returns a new array with all the items, as shown in the following example:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var pizzaVegetableParts = ['pepper', 'onion'];
var pizzaParts = pizzaMeatParts.concat(pizzaVegetableParts);
```

- **indexOf** Locates the item in the array and returns its index, as shown in the following example, in which the *baconIndex* variable will be set to 2:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var baconIndex = pizzaMeatParts.indexOf('bacon');
```

- **join** Creates a string from the items in the array. The items are comma-delimited by default, but you can pass an alternate separator. The following assigns a string containing ‘pepperoni, ham, bacon’ to the *meatParts* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var meatParts = pizzaMeatParts.join();
```

- **lastIndexOf** Searches from the end of the array for the last item in the array that meets the search criteria and returns its index, as shown in the following example, in which the *lastHamIndex* variable will be set to 3:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'ham', 'prosciutto'];
var lastHamIndex = pizzaMeatParts.lastIndexOf('ham');
```

- **pop** Removes and returns the last element of the array. This reduces the length of the array by one. The following example assigns ‘bacon’ to the *lastItem* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var lastItem = pizzaMeatParts.pop();
```

- **push** Adds a new item to the end of an array and returns the new length, as shown in the following example, in which ‘prosciutto’ is added to the end of the array and 4 is assigned to the *newLength* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var newLength = pizzaMeatParts.push('prosciutto');
```

- **reverse** Reverses the order of the items in an array and returns a reference (not a new array) to the reversed array, so the original array is modified. The following example reverses the order of the array:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'prosciutto'];
pizzaMeatParts.reverse();
```

- **shift** Removes and returns the first item in the array. If no items are in the array, the return value is undefined. The following example removes ‘pepperoni’ from the array and assigns it to the *firstItem* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var firstItem = pizzaMeatParts.shift();
```

- **slice** Returns a new array that represents part of the existing array. The slice method has two parameters: *start* and *end*. The start parameter is the index of the first item to include in the result. The end parameter is the index of the item that you don't want included in the result. In the following example, the *mySlice* variable will be assigned 'ham' and 'bacon'. Note that 'meatball' is not included in the result, and the original array is not changed:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'meatball', 'prosciutto'];
var mySlice = pizzaMeatParts.slice(1,3);
```

- **sort** Sorts the items in an array and returns a reference to the array. The original array is modified. The following example sorts the array. After sorting, pizzaMeatParts will contain 'bacon', 'ham', 'meatball', 'pepperoni', 'prosciutto':

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'meatball', 'prosciutto'];
pizzaMeatParts.sort();
```

- **splice** Adds and removes items from an array and returns the removed items. The original array is modified to contain the result. The splice method's first parameter is the starting index of where to start adding or deleting. The second parameter indicates how many items to remove. If 0 is passed as the second parameter, no items are removed. If the second parameter is larger than the quantity of items available for removal, all items from the starting index to the end of the array are removed. After the first two parameters, you can specify as many items as you want to add. The following example removes 'ham' and 'bacon' from the original array and assigns 'ham' and 'bacon' to *mySlice*. In addition, 'spam' is inserted in pizzaMeatParts, which results in pizzaMeatParts containing 'pepperoni', 'spam', 'meatball', 'prosciutto':

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon', 'meatball', 'prosciutto'];
var mySlice = pizzaMeatParts.splice(1,2,'spam');
```

- **toString** All objects have a toString method. For the Array object, toString creates a string from the items in the array. The items are comma-delimited, but if you want a different delimiter, you can use the join method and specify an alternate separator. The following assigns a string containing ‘pepperoni,ham,bacon’ to the *meatParts* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var meatParts = pizzaMeatParts.toString();
```

- **unshift** Adds a new item to the beginning of an array and returns the new length, as shown in the following example, in which ‘prosciutto’ is added to the beginning of the array and 4 is assigned to the *newLength* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var newLength = pizzaMeatParts.unshift('prosciutto');
```

- **valueOf** All objects have a **valueOf** method. For the Array object, **valueOf** returns the primitive values of the array as a comma-delimited string, as shown in the following example, which assigns a string containing ‘pepperoni,ham,bacon’ to the *meatParts* variable:

```
var pizzaMeatParts = ['pepperoni', 'ham', 'bacon'];
var meatParts = pizzaMeatParts.valueOf();
```

Remarks:

1. The methods **pop** and **shift** are similar, except that **pop** operates on the end of the array and **shift** on the beginning.
2. The methods **push** and **unshift** are similar, except that **push** operates on the end of the array and **unshift** on the beginning.

Array Iteration Methods

- **forEach** The forEach() method calls a function (a callback function) once for each array element.

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction);

function myFunction(value, index, array) {
    txt = txt + value + "<br>";
}
```

or

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction);

function myFunction(value) {
    txt = txt + value + "<br>";
}
```

- **map** The map() method creates a new array by performing a function on each array element. The map() method does not execute the function for array elements without values. The map() method does not change the original array.

```
var numbers1 = [45, 4, 9, 16, 25];
var numbers2 = numbers1.map(myFunction);

function myFunction(value, index, array) {
    return value * 2;
}
```

or

```
var numbers1 = [45, 4, 9, 16, 25];
var numbers2 = numbers1.map(myFunction);

function myFunction(value) {
    return value * 2;
}
```

- **filter** The filter() method creates a new array with array elements that passes a test.

```
var numbers = [45, 4, 9, 16, 25];
var over18 = numbers.filter(myFunction);

function myFunction(value, index, array) {
    return value > 18;
}
```

or

```
var numbers = [45, 4, 9, 16, 25];
var over18 = numbers.filter(myFunction);

function myFunction(value) {
    return value > 18;
}
```

- **every** The every() method check if all array values pass a test. It returns a boolean value.

```
var numbers = [45, 4, 9, 16, 25];
var allOver18 = numbers.every(myFunction);

function myFunction(value, index, array) {
    return value > 18;
}
```

or

```
var numbers = [45, 4, 9, 16, 25];
var allOver18 = numbers.every(myFunction);

function myFunction(value) {
    return value > 18;
}
```

- **some** The some() method check if some array values pass a test. It returns a boolean value.

```
var numbers = [45, 4, 9, 16, 25];
var someOver18 = numbers.some(myFunction);

function myFunction(value, index, array) {
    return value > 18;
}
```

or

```
var numbers = [45, 4, 9, 16, 25];
var someOver18 = numbers.some(myFunction);

function myFunction(value) {
    return value > 18;
}
```

There are other iteration methods such as reduce(), reduceRight(), find() and findIndex().

Accessing DOM objects

When building an application, the primary objects you must access are the objects that make up the DOM, which represents the HTML document. You need to access the DOM to control the behavior of your HTML document and to be notified when something happens on the page.

Navigating the DOM

The DOM represents a hierarchy of objects, forming a model of your HTML document. To retrieve elements from the DOM, use the built-in *document* variable, which references the DOM, and perform one of the search methods. Some of the search methods return a single element, whereas others return a NodeList of elements.

NodeLists are very similar to arrays. They are manipulated essentially in the same way, but there are some array methods that do not apply to NodeLists. If necessary, a NodeList can be converted into an array by the function `Array.from(nl)`, where `nl` is the NodeList. (See Remark 2 at the end of this section.)

The methods that return a NodeList return either a *live NodeList* or a *static NodeList*. The live NodeList represents a collection of elements that is continuously updated as the DOM changes, whereas the static NodeList represents a snapshot of elements that doesn't change as the DOM changes. From a performance perspective, it takes longer to create the static NodeList, so consider working with the search methods that return a live NodeList if you want the best performance. It's important to understand this difference because it can affect your choice of search method. The following is a list of the DOM search methods with a short description and example:

getElementById Returns a reference to the first object with specified id, as shown in the following example, which retrieves a reference to the button with the id of btnSave:

```
var btn = document.getElementById('btnSave');
```

getElementsByTagName Returns a live NodeList, which is a special array of all elements with the specified tag name. The live NodeList automatically updates if you add, delete, or modify elements. In addition to being supported on the document object, the getElementsByTagName method exists on the Element object, so you can query either the entire DOM or just an element's content. The following example returns an array of all images:

```
var images = document.getElementsByTagName('img');
```

getElementsByName Returns a live NodeList of all elements with the specified name. This works well with option buttons when all their options typically have the same name. The following example retrieves an array of all elements with the name pizzaSize:

```
var pizzaSizes = document.getElementsByName('pizzaSize');
```

getElementsByClassName Not supported in Internet Explorer 8 and earlier. Returns a live NodeList of all elements with the specified CSS class name. In addition to being supported on the document object, the getElementsByClassName method exists on the Element object, so you can query either the entire DOM or just an element's content. The following example retrieves an array of all elements with the class name pizzaPart:

```
var pizzaParts= document.getElementsByClassName('pizzaPart');
```

querySelector Not supported in Internet Explorer 7 and earlier. Accepts a CSS selector as its parameter. The querySelector method returns the first matched element if one or many exist or null if there is no match. In addition to being supported on the document object, the querySelector method exists on the Element object, so you can query either the entire DOM or just an element's content. In the following example, the pound symbol (#) indicates a search for an id. This example returns a reference to the button whose id is btnSave:

```
var btn = document.querySelector('#btnSave');
```

querySelectorAll Not supported on Internet Explorer 7 and earlier. Accepts a CSS selector as its parameter. The querySelectorAll method returns a static NodeList of all elements that match or an empty array if there is no match. In addition to being supported on the document object, the querySelectorAll method exists on the Element object, so you can query either the entire DOM or just an element's content. In the following example, the period (.) indicates a search for a CSS class name. This example returns a reference to the elements whose CSS class name is pizzaPart:

```
var btn = document.querySelectorAll('.pizzaPart');
```

Remarks:

1. An example combining Document methods and Element methods:

```
<div id="div02">
    <p>Good Morning</p>
</div>
<script>
    document.getElementById("div02").getElementsByTagName("p")[0].innerHTML += ", Hello";
</script>
```

The result will correspond to `<p>Good Morning, Hello</p>`.

2. An example showing how to use Array.from() on a NodeList. Then method `forEach` is not supported in NodeList (NodeLists are not really arrays), that is why we need the function `Array.from()` before applying `forEach`.

```
<div id="div03">
    <p>Good Morning</p>
    <p>Hello</p>
</div>
<script>
    var x = document.getElementById("div03").getElementsByTagName("p");
    (Array.from(x)).forEach(function (value) { value.innerHTML += " !"; });
</script>
```

The result will correspond to:

```
<p>Good Morning !</p>
<p>Hello !</p>
```

References:

Johnson, Glenn - *Programmingin HTML5 with JavaScript and CSS3 – Training Guide*, Microsoft Press, 2013

Haverbeke, Marijn - *Eloquent JavaScript* – 3rd Edition, No Starch Press, 2018

Patel, Meher Krishna - *HTML, CSS, Bootstrap, Javascript and jQuery* – PythonDSP, October 2018

http://eloquentjavascript.net/

https://www.w3schools.com/js