Arrays - Notes

The array is the most common data structure in computer programming. Every programming language includes some form of array. Because arrays are built-in, they are usually very efficient and are considered good choices for many data storage purposes. In this chapter we explore how arrays work in JavaScript and when to use them.

JavaScript Arrays Defined

The standard definition for an array is a linear collection of elements, where the elements can be accessed via indices, which are usually integers used to compute offsets. Most computer programming languages have these types of arrays. JavaScript, on the other hand, has a different type of array altogether.

A JavaScript array is actually a specialized type of JavaScript object, with the indices being property names that can be integers used to represent offsets. However, when integers are used for indices, they are converted to strings internally in order to conform to the requirements for JavaScript objects. Because JavaScript arrays are just objects, they are not quite as efficient as the arrays of other programming languages.
While JavaScript arrays are, strictly speaking, JavaScript objects, they are specialized objects categorized internally as arrays. The Array is one of the recognized JavaScript object types, and as such, there is a set of properties and functions you can use with arrays.

Arrays in JavaScript are very flexible. There are several different ways to create arrays, access array elements, and perform tasks such as searching and sorting the elements stored in an array.

Creating Arrays

The simplest way to create an array is by declaring an array variable using the [] operator:

```
var numbers = [];
```

When you create an array in this manner, you have an array with length of 0. You can verify this by calling the built-in length property:

```
var numbers = [];
```

Another way to create an array is to declare an array variable with a set of elements inside the [] operator:

```
var numbers = [1,2,3,4,5];
print(numbers. length); // 5
```

You can also create an array by calling the Array constructor:

```
var numbers = new Array();
print(numbers.length); // displays 0
```

You can call the Array constructor with a set of elements as arguments to the constructor:

```
var numbers = new Array(1,2,3,4,5);
print(numbers.length); // displays 5
```

Finally, you can create an array by calling the Array constructor with a single argument specifying the length of the array:

```
var numbers = new Array(10);
print(numbers.length); // displays 10
```

Unlike many other programming languages, but common for most scripting languages, JavaScript array elements do not all have to be of the same type:

```
var objects = [1, "Joe", true, null];
```

We can verify that an object is an array by calling the Array.isArray() function, like this:

```
var nums = [];
for(vari=0;i<100;++i) {nums[i] = i+1;}
```

We've covered several techniques for creating arrays. As for which function is best, most JavaScript experts recommend using the [] operator, saying it is more efficient than calling the Array constructor

Accessing and Writing Array Elements

Data is assigned to array elements using the [] operator in an assignment statement. For example, the following loop assigns the values 1 through 100 to an array:

```
var nums = [];
for(vari=0;i<var nums = [];
for(vari=0;i<100;++i) {nums[i] = i+1;};++i) {nums[i] = i+1;}
```

Array elements are also accessed using the [] operator. For example:

```
var numbers = [1,2,3,4,5];
var sum = numbers[0] + numbers[1] + numbers[2] + numbers[3] +numbers[4];
print (sum); // displays 15
```

Of course, accessing all the elements of an array sequentially is much easier using a for loop:

```
var numbers = [1,2,3,5,8,13,21];
var sum=0;
for (var i = 0; i < numbers.length; ++i{
sum += numbers[i];
}
```

```
print(sum); // displays 53
```

Notice that the for loop is controlled using the length property rather than an integer literal. Because JavaScript arrays are objects, they can grow beyond the size specified when they were created. By using the length property, which returns the number of elements currently in the array, you can guarantee that your loop processes all array elements.

Adding Elements to an Array

There are two mutator functions for adding elements to an array: push() and un shift(). The push() function adds an element to the end of an array:

```
var nums = [1,2,3,4,5]; print(nums); // 1,2,3,4,5
nums.push(6);
print(nums); // 1,2,3,4,5,6
```

Using push() is more intuitive than using the length property to extend an array:

```
var nums = [1,2,3,4,5];
print(nums); // 1,2,3,4,5
nums[nums.length] = 6;
```

Adding data to the beginning of an array is much harder than adding data to the end of an array. To do so without the benefit of a mutator function, each existing element of the array has to be shifted up one position before the new data is added. Here is some code to illustrate this scenario:

```
var nums = [2,3,4,5];
var newnum = 1;
var N = nums.length; for(vari=N;i>=0;--i){
nums[i] = nums[i-1];}
nums[0] = newnum; print(nums); // 1,2,3,4,5
```

This code becomes more inefficient as the number of elements stored in the array increases.

The mutator function for adding array elements to the beginning of an array is un shift(). Here is how the function works:

```
var nums = [2,3,4,5];
print(nums); // 2,3,4,5
var newnum = 1;
nums.unshift(newnum);
print(nums); // 1,2,3,4,5
nums = [3,4,5];
nums.unshift(newnum,1,2);
print(nums); // 1,2,3,4,5
```

The second call to unshift() demonstrates that you can add multiple elements to an array with one call to the function.

Removing Elements from an Array

Removing an element from the end of an array is easy using the pop() mutator function:

```
var nums = [1,2,3,4,5,9]; nums.pop();
print(nums); // 1,2,3,4,5
```

Without mutator functions, removing elements from the beginning of an array requires shifting elements toward the beginning of the array, causing the same inefficiency we see when adding elements to the beginning of an array:

```
var nums = [9,1,2,3,4,5];
print(nums);
for (var i = 0; i < nums.length; ++i) {
      nums[i] = nums[i+1];
}
print(nums); // 1,2,3,4,5,
```

Besides the fact that we have to shift the elements down to collapse the array, we are also left with an extra element. We know this because of the extra comma we see when we display the array contents.

The mutator function we need to remove an element from the beginning of an array is shift(). Here is how the function works:

```
var nums = [9,1,2,3,4,5]; nums.shift(); print(nums); // 1,2,3,4,5
```

You'll notice there are no extra elements left at the end of the array. Both pop() and shift() return the values they remove, so you can collect the values in a variable:

```
var nums = [6,1,2,3,4,5];
var first = nums.shift(); // first gets the value 9 nums.push(first);
print(nums); // 1,2,3,4,5,6
```

Adding and Removing Elements from the Middle of an Array

Trying to add or remove elements at the end of an array leads to the same problems we find when trying to add or remove elements from the beginning of an array—both operations require shifting array elements either toward the beginning or toward the end of the array. However, there is one mutator function we can use to perform both operations—splice().

To add elements to an array using splice(), you have to provide the following arguments:

The starting index (where you want to begin adding elements)
The number of elements to remove (0 when you are adding elements)
The elements you want to add to the array
Let's look at a simple example.
The following program adds elements to the middle of an array:

```
var nums = [1,2,3,7,8,9];
var newElements = [4,5,6];
nums.splice(3,0,newElements);
print(nums); // 1,2,3,4,5,6,7,8,9
```

The elements spliced into an array can be any list of items passed to the function, not necessarily a named array of items. For example:

```
var nums = [1,2,3,7,8,9];
nums.splice(3,0,4,5,6);
print(nums);
```

In the preceding example, the arguments 4, 5, and 6 represent the list of elements we want to insert into nums.
Here is an example of using splice() to remove elements from an array:

```
var nums = [1,2,3,100,200,300,400,4,5];
nums.splice(3,4);
print(nums); // 1,2,3,4,5
```

Putting Array Elements in Order

The last two mutator functions are used to arrange array elements into some type of order. The first of these, reverse(), reverses the order of the elements of an array. Here is an example of its use:

```
var nums = [1,2,3,4,5];
nums.reverse();
print(nums); // 5,4,3,2,1
```

We often need to sort the elements of an array into order. The mutator function for this task, sort(), works very well with strings:

```
var names = ["David","Mike","Cynthia","Clayton","Bryan","Raymond"];
nums.sort();
print(nums); // Bryan,Clayton,Cynthia,David,Mike,Raymond
```

But sort() does not work so well with numbers:

```
var nums = [3,1,2,100,4,200]; nums.sort();
print(nums); // 1,100,2,200,3,4
```

The sort() function sorts data lexicographically, assuming the data elements are strings, even though in the preceding example, the elements are numbers. We can make the sort() function work correctly for numbers by passing in an ordering function as the first argument to the function, which sort() will then use to sort the array elements. This is the function that sort() will use when comparing pairs of array elements to determine their correct order.

For numbers, the ordering function can simply subtract one number from another number. If the number returned is negative, the left operand is less than the right operand; if the number returned is zero, the left operand is equal to the right operand; and if the number returned is positive, the left operand is greater than the right operand.

With this in mind, let's rerun the previous small program using an ordering function:

```
function compare(num1, num2) { return num1 - num2; }
var nums = [3,1,2,100,4,200];
nums.sort(compare);
print(nums); // 1,2,3,4,100,200
```

The sort() function uses the compare() function to sort the array elements numerically rather than lexicographically.

Searching for a Value

One of the most commonly used accessor functions is indexOf(), which looks to see if the argument passed to the function is found in the array. If the argument is contained in the array, the function returns the index position of the argument. If the argument is not found in the array, the function returns -1. Here is an example:

```
var names = ["David", "Cynthia", "Raymond", "Clayton", "Jennifer"];
putstr("Enter a name to search for: ");
var name = readline();
var position = names.indexOf(name);
if (position >= 0) {
```

```
print("Found " + name + " at position " + position);
} else {
print(name + " not found in array.");
}
```

If you run this program and enter **Cynthia**, the program will output: Found Cynthia at position 1

If you enter **Joe**, the output is: Joe not found **in** array.

If you have multiple occurrences of the same data in an array, the indexOf() function will always return the position of the first occurrence. A similar function, lastIndex Of(), will return the position of the last occurrence of the argument in the array, or -1 if the argument isn't found. Here is an example:

```
var names = ["David", "Mike", "Cynthia", "Raymond", "Clayton", "Mike",
"Jennifer"];      var name = "Mike";
var firstPos = names.indexOf(name);
print("First found " + name + " at position " + firstPos); var lastPos =
names.lastIndexOf(name);
print("Last found " + name + " at position " + lastPos);
```

The output from this program is:

```
First found Mike at position 1
Last found Mike at position 5
```

String Representations of Arrays

There are two functions that return string representations of an array: join() and toString(). Both functions return a string containing the elements of the array delimited by commas. Here are some examples:

```
var names = ["David", "Cynthia", "Raymond", "Clayton", "Mike", "Jennifer"];
var namestr = names.join();
print(namestr); // David,Cynthia,Raymond,Clayton,Mike,Jennifer
namestr = names.toString();
print(namestr); // David,Cynthia,Raymond,Clayton,Mike,Jennifer
```