Array methods: splice(), push(), pop(), shift(), unshift()

A versatile splice() method

Inserts new, deletes existing, and replaces existing elements with new elements in the array. The beauty of splice() is that it does not leave the array sparse because it will shift the elements to higher or lower positions as necessary; there will be no gap left.

The first argument of splice() specifies the ordinal position at which the operation is to start. The required second argument specifies the number of elements to delete. The method operates on the array at hand and the return value consists of the array elements deleted.

In the following example the first splice indicates to begin the operation at index 5 and length 2. When the operation completes, the elements f and g have been removed. The second splice indicates a starting position of 2 and length 1. After the operation completes, the letters array has these elements left: a, b, e.

1. var letters = ["a","b","c","d","e","f","g"];
2. alert(letters.splice(5, 2)); // => f, g (deleted from 6th 2 elements)
3. alert(letters); // => a, b, c, d, e
4. alert(letters.splice(2, 1)); // => c (the deleted element)
5. alert(letters); // => a, b, d, e

Using the third argument and higher, the splice method can also be used **to replace one or more elements** with others. In the example below, the splice

* Starts at position 1
* Deletes two elements.
* Replaces the gap with the three elements provided: X, Y, Z.

1. var letters = ["a","b","c","d"];
2. alert(letters.splice(1, 2, "X", "Y", "Z")); // => b, c (deleted ones)
3. alert(letters); // => a, X, Y, Z, d

You can also use splice() to inject new elements into an array without deleting existing ones. Simply specify a 0 for the second argument, like so:

1. var letters = ["a","b","e"];
2. alert(letters.splice(2, 0, "c", "d")); // => no elements returned
3. alert(letters); // => a,b,c,d,e

Implementing a LIFO stack with push() and pop()

The built-in push() array method appends one or more elements to the end of an array, increments the length as appropriate, and returns the length of the modified array. The built-in pop() array method does the opposite; it deletes the last element, reduces the length by one, and returns the deleted element.

1. var days = ["Monday"];
2. alert(days.push("Tuesday", "Wednesday")); // => 3
3. alert(days); // => Monday, Tuesday, Wednesday
4. alert(days.pop()); // => Wednesday
5. alert(days); // => Monday, Tuesday
6. alert(days.push("Wednesday")); // => 3
7. alert(days)); // => Monday, Tuesday, Wednesday

This code shows that the array methods push() and pop() make it very easy to build a LIFO (last in, first out) stack with JavaScript arrays.

Implementing a FIFO queue with unshift() and shift()

The built-in array method unshift() inserts one or more elements to the beginning of an array, shifts the existing elements up to higher indexes to make space for the new elements, increments the length by the number of elements inserted, and returns the new length. The shift() removes the first element, shifts all the elements with higher indexes down one position to fill the gap, reduces the length by one, and returns value of the element that it removed.

1. var days = ["Wednesday"];
2. alert(days.unshift("Monday", "Tuesday")); // => 3
3. alert(days) // => Monday, Tuesday, Wednesday
4. alert(days.shift()); // => Monday
5. alert(days.shift()); // => Tuesday
6. alert(days); // => Wednesday

[Run](javascript:void(0);)

As you can see, the unshift() and shift() methods make it very easy to build a FIFO (first in, first out) queue with JavaScript arrays.

Manipulating arrays with map(), filter(), reduce()

Three other built-in array methods are map(), filter(), and reduce(). We will discuss each of these.

Transforming array elements with map()

The array method map() is used to change each element in the array and return a new array with the modified elements. For example, you could use map to iterate over an array with numbers and then create a new array in which each element is double the value of the original array. The array returned is of the same length as the original array.

Each element of the array on which map() is invoked is passed to a callback function you pass into to map() as an argument. The values returned by the callback are returned in a new array, which is called double in our example below. The original array with the name values on which map() is invoked, is not modified. Since an array is an object, you can also add function methods directly in an array itself.

1. var values = [1, 2, 3, 4, 5];
2. alert(values.length); // => 5
3. var arr2 = values.map( function (value) {
4. return 2 \* value;
5. });
7. alert(arr2); // => 2, 4, 6, 8, 10 (NEW ARRAY
8. alert(arr2.length); // => 5

If the values array were a sparse array with missing values, then the double array would also be sparse with gaps in the elements. Both would still be of the same length.

Filtering the array using filter()

Array method reduce() is an accumulator method that iterates over the elements of the array, from left to right, and reduce it to a single value.

The first argument in reduce() is the callback function and the second optional argument is the initial value. The callback function accepts 4 arguments, of which the first 2 are the previous value and the current value. This is best explained with an example.

Say we have a numeric array and we wish to get the sum of all elements. This is how you would do it with reduce():

1. var values = [2, 4, 6]
2. var sum = values.reduce(function(prev, curr) {
3. return prev + curr;
4. },
5. 0 // initial value
6. );
7. alert(sum); // => 12