Most introductions to JavaScript start with basic **for** loops. This is poor advice. Many of the loops you write are iterations over arrays. These can be done better using array functions specifically designed for that purpose.

map

If **lst** is an array, then **lst.map(***function***)** calls the function for every element in the array. It passes the function the element and the index (zero-based) of that element. The result of the **map()** call is a list of the values returned by the function.

So, if here's a simple loop to take an array of numbers and return a list of the squares of those numbers. Try this in your browser's JavaScript Console.

```
[1, 2, 3, 4, 5].map(function(x) { return x*x; });
```

Our function only needs the first argument.

A more realistic case might be the following: you have a list of names and you want a numbered list of those names, to display in HTML. Let's represent a numbered with an object of the form `{ index: n, name: name }`. Let's assume you want the numbering to start at 1. The following simple **map()** will create your list.

```
["John", "Mary", "Ben"].map(function(x, i) { return { index: i + 1, name: x }; });
```

You can then feed the result to a template for display in HTML.

forEach

lst.forEach(*function***)** works just like **map()** except that the return value is undefined. You use **forEach()** for side effect, e.g., logging every element to the console, or modifying a set of HTML elements.

filter

The **filter(***function***)** lets you take an array and collect just the elements you want. It calls the function for every element. If the function returns true, the element is collected. If the function returns false, the element is skipped.

For example, suppose you had an array of numbers, positive and negative. This code will return a list of all the non-negative numbers.

```
[5, -2, 8, 7, -10, -3, 4].filter(function(x) { return x >= 0; });
```

some and every

The array functions **some(***function***)** and **every(***function***)** let you test an array to see if any or all the elements have some property.

some(*function***)** calls the function for every element of the array. If the function returns true for any element, **some()** returns true. Otherwise, **some()** returns false.

So the following code will return true if any element is negative:

```
[5, -2, 8, 7, -10, -3, 4].some(function(x) { return x < 0; });
```

some() goes left to right. As soon as the function is true, it stops and returns true.

every(*function***)** calls the function for every element of the array. If the function returns true for every element, **every()** returns true. Otherwise, **every()** returns false.

So the following code will return only if all elements are negative:

```
[5, -2, 8, 7, -10, -3, 4].every(function(x) { return x < 0; });
```

reduce

*lst***.reduce(***function***,** *initial-value***)** is the most complicated of the array mapping functions, but the most powerful as well. It lets you loop over an array and accumulate calculations into a final value, any way you want. It takes one *function* and an *initial-value*. For each element of the array *lst*, it calls *function* with three arguments:

- the *accumulator*, which holds the result so far
- the current element
- the current element's index, zero-based

When the loop finishes, the final value of the *accumulator* is returned.

The value of the *accumulator* is

- *initial-value* for the first element
- the value returned by *function* for the previous element, for the remaining elements:

TIP:
Be sure your function always returns the value you want to pass to the next iteration of the loop.

To see how to use **reduce()** for common problems, here are some examples. Try them out in your JavaScript Console.

Suppose we wanted to add up the numbers in an array. Clearly we want to add each element to the accumulator. To start, the accumulator should be zero. So, here's the code:

```
[1, 2, 3, 4, 5].reduce(function(v, x) { return v + x; }, 0);
```

To multiply all the numbers in an array, clearly we should change addition to multiplication.

```
[1, 2, 3, 4, 5].reduce(function(v, x) { return v * x; }, 0);  // wrong!
```

Ooops! This always returns zero! This is obvious in hindsight. 0 times 1 times 2 ... is zero. The correct initial value is one. One is the *multiplicative identity*, just as zero is the *additive identity*. So the correct code for multiplying numbers in a list is:

```
[1, 2, 3, 4, 5].reduce(function(v, x) { return v * x; }, 1);
```

The above code is easy to modify if we want to apply some function to every element of an array and sum or multiply the results. For example, to get the sum of the squares of every element of an array, just modify the additive loop:

```
[1, 2, 3, 4, 5].reduce(function(v, x) { return v + x * x; }, 0);
```

The power of **reduce()** comes from the fact that we can do anything we want to calculate the new value for the accumulator. For example, suppose we wanted to add only the even numbers. The standard way to test for even numbers in JavaScript is to use the *modulus* operator, i.e., **x % 2** returns the remainder of **x** divided by 2, so **x % 2 === 0** tests to see if that remainder is 0, i.e., **x** is even.

To add the even numbers in a list, we need a function that returns the current sum + x, if x is even. If x is not even, we want to return the sum unchanged. We could write that with an **if**:

```
[1, 2, 3, 4, 5].reduce(function(v, x) { if (x % 2 === 0) { return v + x; } else { return v; } }, 0);
```

A shorter way, with a lot less punctuation, is to use the **?:** operator pair:

```
[1, 2, 3, 4, 5].reduce(function(v, x) { return (x % 2 === 0) ? v + x : v }, 0);
```

join

The array function **join()** is used to combine[all the elements of a list into a single string. This is most useful when building text or HTML for a web page. **join()** takes one parameter, which should be the string to use in between the joined elements. For example

```
[1, 2, 3].join(",")
```

returns the string "1,2,3". If you don't want any separator, use the empty string:

```
["a","e","i","o","u"].join("")
```

returns the string "aeiou".

Combining **join()** with **map()** and/or **filter()** can construct complex HTML with very code, e.g., the code

```
var attendees = [{ name: "John", state: "Iowa" },
  { name: "Mary", state: "Illinois" },
  { name: "Alvin", state: "Indiana" }
];

var htmlList = attendees.map(function(attendee) {
  return "<li>" + attendee.name + " from " + attendee.state + "</li">;
}).join("");
```

would create a string with HTML list elements for an event.