A Drip of JavaScript

# Boiling Down Arrays with Array#reduce

*Originally published in the A Drip of JavaScript newsletter.*

We've talked before about the advantages of using `Array`'s higher order functions (like `map` and `sort`) rather than manually iterating over an array's contents. But unlike `map` and `sort`, which produce new arrays, `reduce` can boil an array down to a new value of any type.

Consider the case of a library system which needs to show potential donors its effectiveness at spreading knowledge. In order to do that, it must determine how many book-checkouts there were last year. (Not total of all checkouts, since a single checkout may be for more than one book.)

Their data is stored in an object like this:

```
var books = [
    {
        title: "Showings",
        author: "Julian of Norwich",
        checkouts: 45
    },
    {
        title: "The Triads",
        author: "Gregory Palamas",
        checkouts: 32
    },
    {
```

```
        title: "The Praktikos",
        author: "Evagrius Ponticus",
        checkouts: 29
    }
];
```

The traditional C-style way to get at that number would be something like this:

```
var bookCheckouts = 0;

for (var i = 0; i < books.length; i++) {
    bookCheckouts = bookCheckouts + books[i].checkouts;
}
```

But using `reduce`, we can think more directly about the problem itself:

```
// Get an array of checkout values only
var bookCheckouts = books.map(function(item) {
    return item.checkouts;
});

// Sum the array's values from left to right
var total = bookCheckouts.reduce(function(prev, curr) {
    return prev + curr;
});
```

While it may look a little foreign at first, this example is more explicit about the problem than the `for` version is. Namely, dealing only with the `checkouts` property, and totaling them. And if you are familiar with the idioms, you can make it almost as concise.

```
// Get total of book checkouts
var total = books
            .map(function(b) { return b.checkouts; })
            .reduce(function(p, c) { return p + c; });
```

So how does that `reduce` method work? It takes an array of values and executes the function (almost!) once for each item in the array. The value of `prev` is the result of the calculations up to this point. Let's take a look:

```
[1, 2, 3].reduce(function(prev, curr, index) {
    console.log(prev, curr, index);
    return prev + curr;
});

// Outputs:
// 1, 2, 1
// 3, 3, 2
// 6
```

As you can see, the callback actually didn't iterate over the first element of the array. Instead, `prev` was set to the value of the first element. It has to work that way because otherwise `reduce` wouldn't have a starting value for `prev`.

The function ran three times, but on the last call, it wasn't given a current element since there were no elements left in the array. That behavior is a little confusing to read through, but is quite easy to use.

Here is another example:

```
[1, 2, 3].reduce(function(prev, curr, index) {
    console.log(prev, curr, index);
    return prev + curr;
}, 100);

// Outputs:
// 100 1 0
// 101 2 1
// 103 3 2
// 106
```

In addition to the callback function, we also passed in an initial value for `prev`. You'll note that in this case, the callback **did** iterate over the first element in the array, since it had an initial value to work with.

It should be noted that your callback function also has access to the array itself as a parameter.

```
[1, 2, 3].reduce(function(prev, curr, index, arr) {
    // Do something here
});
```

Is reduce only good for working with numbers? By no means. Consider this:

```
var relArray = [
    ["Viola", "Orsino"],
    ["Orsino", "Olivia"],
    ["Olivia", "Cesario"]
];

var relMap = relArray.reduce(function(memo, curr) {
    memo[curr[0]] = curr[1];
    return memo;
}, {});

// Outputs: {
//    "Viola": "Orsino",
//    "Orsino": "Olivia",
//    "Olivia": "Cesario"
// }
console.log(relMap);
```

This time we used `reduce` to transform the relationship data from an array of arrays into an object where each key value pair describes the relationship. You'll also see that instead of naming the parameter `prev` as I did before, I used the term `memo`. That's because in this case, `memo` is a more descriptive name for the actual value being passed in (an object collecting new properties.)

The `reduce` method is good for any sort of situation where we want to take an array and boil it down into a new value in steps that have access to one or two of the array elements at a time.

Want to use `reduce` in IE8 or IE7? Take a look at the Underscore and Lo-Dash libraries.

---