A Drip of JavaScript

Using 'apply' to Emulate JavaScript's Upcoming Spread Operator

Originally published in the [A Drip of JavaScript newsletter](#).

The next version of JavaScript, ECMAScript 6, is planned to introduce a handy [spread operator](#) which makes using arrays to supply function arguments much simpler. Here's an example of how it will work:

```
var someArgs = ["a", "b", "c"];

// Using the spread operator with someArgs
console.log(...someArgs);

// Is equivalent to this
console.log("a", "b", "c");
```

Now some of you might be thinking that `apply` gives us a nice easy way to emulate that, and you'd be right. This is also equivalent to the above:

```
console.log.apply(console, someArgs);
```

The `apply` method is part of the prototype of all functions, so any time we have a function, we can invoke it using `apply.` It takes two arguments, a `this` parameter which sets the value of `this` within the function, and a single array which is converted into a list of arguments to the function that `apply` is called on.

So if `apply` gives us the same thing as the spread operator, why is the ES6 committee adding spread at all? Because the spread operator is much more flexible. See for example:

```
var someArgs = ["a", "b", "c"];
var moreArgs = [1, 2, 3];

// Using the spread operator twice, with a non-spread argument
console.log(...someArgs, "between", ...moreArgs);

// Is equivalent to this
console.log("a", "b", "c", "between", 1, 2, 3);
```

In this case, there isn't a simple one to one mapping between using the spread operator and using `apply`. But we can still create a pretty decent emulation. Let's take a look at how.

```
function spreadify (fn, fnThis) {
    return function (/* accepts unlimited arguments */) {
        // Holds the processed arguments for use by `fn`
        var spreadArgs = [ ];

        // Caching length
        var length = arguments.length;

        var currentArg;

        for (var i = 0; i < length; i++) {
            currentArg = arguments[i];

            if (Array.isArray(currentArg)) {
                spreadArgs = spreadArgs.concat(currentArg);
            } else {
                spreadArgs.push(currentArg);
            }
        }

        fn.apply(fnThis, spreadArgs);
    };
}

var someArgs = ["a", "b", "c"];
var moreArgs = [1, 2, 3];

// Outputs: ["a", "b", "c"] [1, 2, 3]
console.log(someArgs, moreArgs);

// Outputs: a b c 1 2 3
spreadify(console.log, console)(someArgs, moreArgs);
```

What on earth is going on here? It's actually pretty simple. Our `spreadify` function takes two arguments, a function, and an optional parameter specifying what the function's `this` should be. In the case of `console.log`, that is `console`.

When the `spreadify` function is invoked, it immediately returns another function. That's why we have the doubled parentheses at the end of the example. But you could also do something like this:

```
// `spreadLog` is now the function that `spreadify` returns
var spreadLog = spreadify(console.log, console);

// Outputs: a b c 1 2 3
spreadLog(someArgs, moreArgs);
```

So we have our new function. That new function does two things.

1. It does some preprocessing on whatever arguments we pass it. (Either immediately with that second set of parentheses in the first example, or later on as in the second example.)

2. It then invokes our original function (`log` in this case) with the processed arguments.

In our preprocessing we use the `arguments` object to examine all the arguments which were passed in. If an argument is an array, then we add the array's contents to `spreadArgs`. If the argument is not an array, then we directly push the argument onto `spreadArgs`.

Finally, we invoke our original function by using `apply`, passing in the `this` value, as well the `spreadArgs` array.

In the event that you didn't pass a `this` value into `spreadify`, `fnThis` will be `undefined`, and `apply` will use the default value of `this` (usually the global object).

You may have noticed one problem with our emulation of the spread operator here. The `spreadify` function assumes that all arrays should be expanded into individual arguments. That may not always be the case. Fortunately there's a fairly simple workaround. Just wrap any array you want to preserve as an individual argument inside another array, like so:

```
var someArgs = ["a", "b", "c"];
var moreArgs = [1, 2, 3];
var myArray = ["my", "array"];

// `spreadLog` is now the function that `spreadify` returns
var spreadLog = spreadify(console.log, console);

// Outputs: a b c 1 2 3 ["my", "array"]
spreadLog(someArgs, moreArgs, [myArray]);
```

Of course, `spreadify` doesn't completely replace the spread operator, since in ES6 you can also spread into an array literal:

```
var alpha = ["a", "b", "c"];
var num = [1, 2, 3];

var all = ["all", ...alpha, ...num];
```

But `spreadify` can still be pretty useful. And hopefully walking through how it uses `apply` has helped give you a better grasp of how powerful JavaScript functions can be.
