A Drip of JavaScript

Invoking JavaScript Functions With 'call' and 'apply'

Originally published in the A Drip of JavaScript newsletter.

A couple of issues ago, we talked about some of the implications of functions being first-class citizens in JavaScript. Here is a further implication to consider: **If functions are objects and objects can have methods, then functions can have methods.**

In fact, JavaScript functions come with several methods built into `Function.prototype`. First let's take a look at `call`.

```
function add (a, b) {
    return a + b;
}

// Outputs: 3
console.log(add(1, 2));

// Outputs: 3
console.log(add.call(this, 1, 2));
```

Assuming that you're not using strict mode, these invocations of `add` are exactly equivalent. The first parameter given to `call` has a special purpose, but any subsequent parameters are treated the same as if `add` had been invoked normally.

The first parameter that `call` expects, though, will be set to `add`'s internal `this` value. When a function is invoked ordinarily, the `this` value is set implicitly.

If you're not in strict mode and the function isn't attached to an object, then it will inherit its `this` from the global object. If the function is attached to an object, its default `this` is the receiver of the method call.

Let's look at how that works:

```
var palestrina = {
    work: "Missa Papae Marcelli",
    describe: function() {
        console.log(this.work);
    }
};

// Outputs: "Missa Papae Marcelli",
palestrina.describe();
```

But `call` gives us a way to "borrow" a method from one object to use for another.

```
var erasmus = {
    work: "Freedom of the Will"
};

// Outputs: "Freedom of the Will"
palestrina.describe.call(erasmus);
```

You may be wondering how this is useful. But we've seen this approach before. Last time, we used it to invoke `Array` methods on a non-array (though array-like) object.

```
function myFunc () {
    // Invoke `slice` with `arguments`
    // as it's `this` value
```

```
    var args = Array.prototype.slice.call(arguments);
}
```

It's use extends beyond the `arguments` object, though. For instance, you can invoke many array methods on strings:

```
var original = "There is 1 number.";

var updated = Array.prototype.filter.call(original, function(val) {
    return val.match(/1/);
});

// Outputs: ["1"]
console.log(updated);

// Outputs: "1"
console.log(updated.join(''));
```

Of course, the return values of those methods will be arrays, so you may need to convert them back to strings with `join`.

So far we've only talked about `call`. So what's the deal with `apply`? It turns out that `apply` works in almost exactly the same way as `call`. The difference is that instead of a series of arguments, `apply` takes an array of values to use in its invocation.

```
function add (a, b) {
    return a + b;
}

// Outputs: 3
console.log(add.call(this, 1, 2));

// Outputs: 3
console.log(add.apply(this, [1, 2]));
```

In the example above, `call` and `apply` are used in exactly equivalent ways. As you can see, the only real difference is that `apply` takes an array.

But that turns out to be a very important difference. Unlike a series of arguments, an array is very easy to manipulate in JavaScript. And that opens up much larger possibilities for working with functions.

In the next issue, we'll explore some of those possibilities.
