A Drip of JavaScript

Function functions

Originally published in the A Drip of JavaScript newsletter.

One of the features of JavaScript that sometimes trips up developers used to other languages is the fact that functions are first-class objects. This means that functions can be assigned to a variable and passed around.

```
var one = function() { return 1; };

// Outputs: "function() { return 1; }"
console.log(one);

// Outputs: 1
console.log(one());
```

Why the difference between the results of these two logs? In the first instance, we are logging the value of `one`. And the value of `one` is the function itself. In the second case we are logging the value returned by invoking `one`.

So far, so good. But there is a further implication. Functions can return objects. And if functions are first-class objects, then that means it is possible for a function to return a function.

```
function outerFunction () {
    return function() { return "inner"; };
}

var whatIsIt = outerFunction();
```

```
// Outputs: function() { return "inner"; }
console.log(whatIsIt);

// Outputs: "inner"
console.log(whatIsIt());
```

This can be a little confusing at first, but is actually pretty simple. When we invoke `outerFunction`, it will immediately return our anonymous inner function. And we can easily assign that to a variable for later use.

Okay. But do functions that return functions have any real use? It turns out that they do. You see, functions created inside another function have a handy property. They hold onto or "close over" the variables that their parent defined.

```
function counterCreator () {
    var count = 0;
    return function() { return ++count; };
}

var counter = counterCreator();

// Outputs: function() { return ++count; }
console.log(counter);

// Outputs: 1
console.log(counter());

// Outputs: 2
console.log(counter());

// Outputs: ReferenceError: count is not defined
console.log(count);
```

Interesting! See how our `counter` function is holding onto the value of `count` even across multiple invocations? And that's despite the fact that the `count`

variable isn't accessible in our top scope.

```
function counterCreator () {
    var count = 0;
    return function() { return ++count; };
}

var originalCounter = counterCreator();
var anotherCounter = counterCreator();

// Outputs: 1
console.log(originalCounter());

// Outputs: 2
console.log(originalCounter());

// Outputs: 1
console.log(anotherCounter());
```

Well, look at that. Each new counter function that we create is able to maintain its own private `count` variable independent of the other counter functions. In fact, this aspect of JavaScript's functions is often used to emulate the sort of private properties found in languages like Java.

That's a little beyond the scope of this issue, but how else can "function functions" be useful? Here's one example:

```
function multiply (a, b) {
    return a * b;
}

function timesCreator (a) {
    return function (b) {
        return multiply(a, b);
    }
}
```

```
timesTwo = timesCreator(2);
timesTwelve = timesCreator(12);

// Outputs: 4
console.log(timesTwo(2));

// Outputs: 24
console.log(timesTwelve(2));
```

This particular approach is called partial application, which is excellent at helping you ensure that there is one canonical location for your core algorithm. When you need more specific variants, you can just generate them automatically.

Or, going back to our example with the counters, perhaps you want to create a function that can be invoked many times, but will only execute it's main logic once.

```
function once (fn) {
    var used = false;
    return function (something) {
        // End function execution if previously invoked
        if (used) { return; }

        // Invoke the original function
        fn(something);

        // Mark the function as used
        used = true;
    }
}

function logStuff (stuff) {
    console.log(stuff);
}

logOnce = once(logStuff);

// Outputs: "One does not simply"
```

```
logOnce("One does not simply");

// No output!
logOnce("use a function twice");
```

Hopefully, this has given you some ideas about how you might use "function functions". We'll be looking more at their practical use in the future. If you're interested in learning more about them, I suggest looking through Underscore's documentation and source code.
