Introduction to Map

Another new method that we got with ECMAScript 5, to the delight of functional programmers everywhere, was `map`. This is another way that you can iterate through the items in an array. Using `map` will look very familiar, since we've just talked about using `forEach`. It operates in the same way, but returns an array of transformed values after it's done operating.

In fact, one of the things you can rely on with `map` is that it's always going to return an array of the same length. That's what makes it convenient for functional programming, where you can chain methods one right after the other and keep performing sequential operations.

```
var steps = ["brainstorm", "narrow", "prototype", "test", "propose"];

var forEachReturns = steps.forEach(function(item) {
  return item;
});

var mapReturns = steps.map(function(item) {
  return item;
});

console.log(forEachReturns);
//undefined
console.log(mapReturns);
//["brainstorm", "narrow", "prototype", "test", "propose"]
```

Notice that `forEach` does not return anything at all, whereas `map` returns an array of items. The next thing you can do with `map` is start chaining operations and performing repeated operations again and again on the same array.

Chaining With Map

That ability to chain with `map` is the foundation of functional programming. In fact, by adding the `map` method to the array, JavaScript turned the array into a functor, which is a functional programming term for a data type that implements the `map` method.

```
var steps = ["brainstorm", "narrow", "prototype", "test", "propose"];

var capitalize = function(word) {
  return word.charAt(0).toUpperCase() + word.slice(1);
};

var appendCount = function(word) {
  return word + " (" + word.length + ")";
};

var mapReturns = steps.map(capitalize).map(appendCount);

console.log(mapReturns);
//["Brainstorm (10)", "Narrow (6)", "Prototype (9)", "Test (4)", "Propose (7)"]
```

`capitalize` takes the first character of the string, turns it to uppercase and then just appends the other symbols in that string. This way we capitalize our string.

Notice how we use `map` here by passing function's name an argument - this function will be called for each item in an array. Remember though that the original array is left unmodified - all operations are carried out on its copy.

Map Can Track Count

Much like `forEach`, `map` also has the ability to keep track of the count while iterating through an array. It's done the same way,b y passing a second argument to the function that's being passed to `map`:

```
var steps = ["brainstorm", "narrow", "prototype", "test", "propose"];

var capitalize = function(word) {
  return word.charAt(0).toUpperCase() + word.slice(1);
};

var appendCount = function(word, count) {
  return word + " (" + count + ")";
};

var mapReturns = steps.map(capitalize).map(appendCount);

console.log(mapReturns);
//["Brainstorm (0)", "Narrow (1)", "Prototype (2)", "Test (3)", "Propose (4)"]
```

`map` has some of the same performance issues that you'll find with `forEach`. JavaScript engines are being improved so they'll probably optimize better to take advantage of `map`. But again, I always recommend developing your code for maintainability and clarity, and then optimizing for performance later when the real world situations demand it.