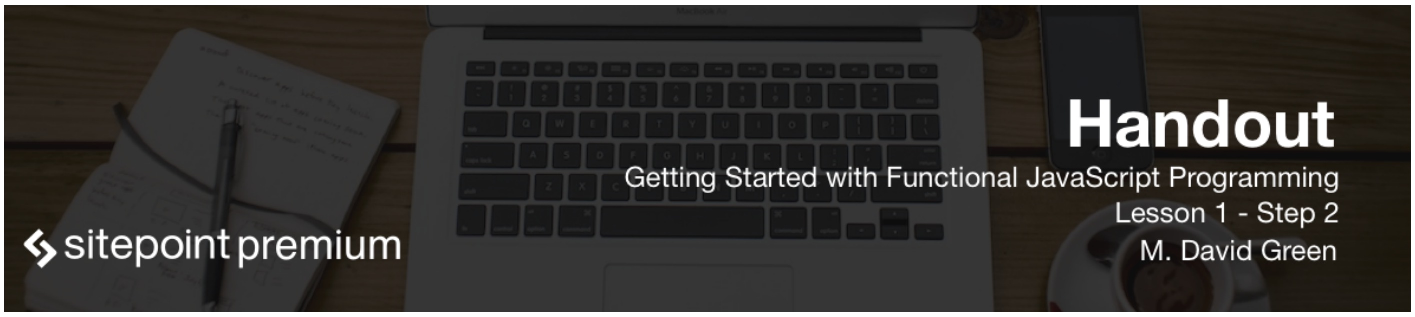

Handout

Getting Started with Functional JavaScript Programming

Lesson 1 - Step 2

M. David Green

To help you understand what functional programming is, why don't we start with imperative JavaScript and take a whirlwind tour through the history of how JavaScript has been done over the years?

Imperative style is code that's read from top to bottom, for the most part. It tells the computer step by step exactly what to do next, and each step depends on knowing exactly where you are in the process. Statements change the state of the application as they're executed, so depending on what you did before, the code you've written may do something completely different this time.

Suppose we have a button with a click event. When it fires, we are asked for a string. All the words in that string will be capitalized and then shown to us.

```
var result;
function getText() {
  var someText = prompt("Give me something to capitalize");
  capWords(someText);
  alert(result.join(" "));
};
function capWords(input) {
  var counter;
  var inputArray = input.split(" ");
  var transformed = "";
  result = [];
  for (counter = 0; counter < inputArray.length; counter++) {
    transformed = [
      inputArray[counter].charAt(0).toUpperCase(),
      inputArray[counter].substring(1)
    ].join("");
    result.push(transformed);
  }
};
document.getElementById("main_button").onclick = getText;
```

If anything about this code looks confusing or unfamiliar to you, you might wanna go back and look at some of the other JavaScript courses, perhaps the Introduction to JavaScript course that SitePoint offers before moving forward with this course. On the other hand if, you saw some things in here that kind of squeaked to you, that you thought maybe you wouldn't do it that way, you're probably ready for this course.

So let's talk about some of the issues that you might have noticed while looking at this imperative style code.

- First of all, we were defining variables in the global scope.

- We also had interdependent functions. In this case, the `capWords` function relied on the `getText` function in order to behave properly.
- We had values being passed around and redefined. So for example, we had the `result` value that was defined, originally, outside of a function and then modified inside of functions.
- We also had native JavaScript in DOM methods mixed together in our code. In this case, we've got the `onclick` method that we used. And it was completely unclear what else might be happening outside of the script that might have affected the values we were working with, or if other functions were relying on the same values that we were modifying.
- We had function names also that might have been repeated in other scripts that could have been loaded in the same context. For example, in a browser, you might have had multiple scripts that all defined a separate `getText` function.
- At the core of it, this code looks kind of brittle. It won't work unless you're in a context in which, for example, you've got a variable called `result` defined outside of our functions. And as programmers were looking at this, and thinking about the versatility of JavaScript, they started thinking about other ways that they could structure their code, so that it would be a little bit more robust.

Let's take a look at one of those.