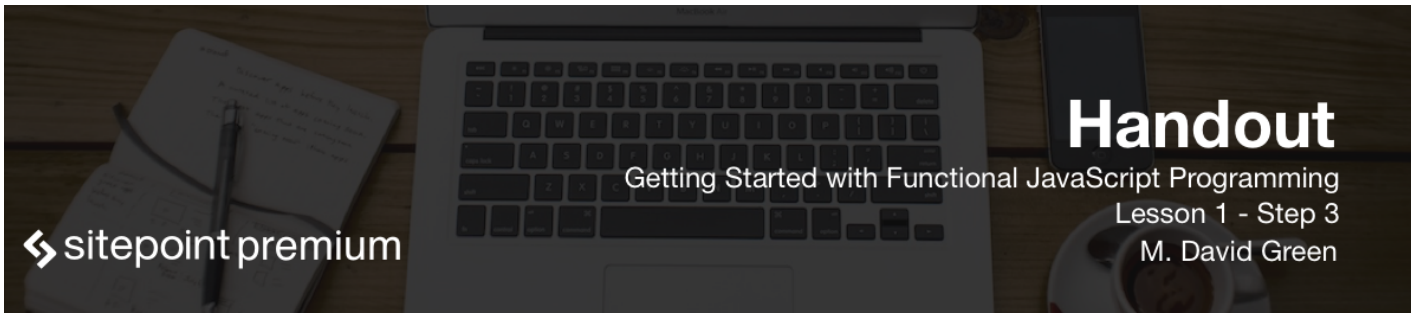

After a while, people started recognizing that there were problems with pure imperative approaches to using JavaScript, especially in an environment where multiple people may be submitting script simultaneously to work in the browser. So people started using JavaScript's **Object Oriented capabilities**. What is Object Oriented Style in terms of JavaScript?

Well, first of all, Object Oriented Style allows the data to be treated as objects to be passed around the code. And you use defined methods to manipulate those objects. Each of those objects can have its own methods and the instances of those objects can also have methods. Messages can be passed around from one object to another and properties of objects can be hidden or shared, allowing for better encapsulation and more security in your code.

Object oriented programming also offers the opportunity to build up new functionality by adding to the abilities of an object through composition. As well as, building up new objects by inheriting properties and capabilities and enhancing them or modifying them.

So let's look at what the code to do exactly what we were trying to do in the previous video would look like using object oriented programming, sometime around 2010.

One of the objectives people were looking at was the importance of keeping your code **encapsulated** so that one script doesn't interfere with another. In order to do that now, why we can create an immediately invoked function expression:

```
(function() {
}());
```

What this does is, creates a safe space in which we can do our coding and everything inside of here won't leak out into the global space.

Another thing that was new around that time was the importance of using **strict**. This allowed us to make sure that the code we were writing followed strict conventions in terms of the use of variables so that we wouldn't be allowed to use a variable without declaring it for example. Let's declare a variable and call it `SomeText`. And I'm gonna capitalize that because this is going to be our object:

```
(function() {
  "use strict";
  var SomeText = function(text) {
    this.text = text;
    };
}());
```

Inside we're creating a local copy of the text that's being passed in so that we have something to work with.

Now, we've got our SomeText function and we're going to build up some methods on top of that:

```
(function() {
  "use strict";
  var SomeText = function(text) {
    this.text = text;
    };

  SomeText.prototype.capify = function(str) {
    var firstLetter = str.charAt(0);
    var remainder = str.substring(1);
    return [firstLetter.toUpperCase(), remainder].join("");
  };
  SomeText.prototype.capifyWords = function() {
    var result = [];
    var textArray = this.text.split(" ");
    for (var counter = 0; counter < textArray.length; counter++) {
      result.push(this.capify(textArray[counter]));
    }
    return result.join(" ");
};
}());
```

Notice that we're declaring these new methods on the prototype of our `SomeText` object. This way, when we create multiple instances, they won't have separate copies in memory of each of these methods.

Now you can see why we needed the text parameter passed into our original SomeText object set to `this.text`. Otherwise this method defined on the prototype wouldn't have access to text.

Instead of adding an `onclick` to the button we're going to do an `addEventListener`:

```
document.getElementById("main_button").addEventListener("click", function(e) {
  var something = prompt("Give me something to capitalize");
  var newText = new SomeText(something);
  alert(newText.capifyWords());
});
```

What's interesting about this is that JavaScript allows us to define our object oriented code which does exactly the same thing, but it looks completely different from our imperative code. You can see some examples where we're using a few of the same pieces of code from one to the other.

But structurally, this is a completely different approach to building your program. And object oriented code has some advantages, and it has some disadvantages. So, let's think about what's better about this than the imperative code that we saw before.

- First of all, we've improved our focus on the encapsulation of functionality. We're keeping the functionality here isolated and separate.
- We've also introduced the ability to introduce "use strict" in a limited scope. Applying use strict at the very top of the code could break other code that doesn't follow strict coding standards. But because we've encapsulated it inside of this immediately invoked function expression, use strict only applies to the code inside of there.

- The methods that we want to use are being defined specific to the type of object that's going to use them which adds some convenience. And we've also defined them on the prototype of that object rather than directly on the instances and that allows us to conserve memory. We only need one copy of those methods and every instance can use them.
- We've used the new addEventListener instead of the onClick and that's more versatile because it allows us to listen for different types of events.

So what could still be improved about this object oriented approach?

- All of the methods we're defining on the prototype of this object and we're doing that because they need access to the variables that we're scoped to this inside of the constructor. That's kind of an awkward construct and it's hard to keep track of while you're doing your development.
- We're also still using looping in order to go through all of the elements. And there are a couple of reasons why looping can be messy. For one thing, creating a loop like this relies on having an extra variable, such as counter. It's good to avoid creating extra state that you have to keep track of if you can, and avoid creating extra variables.
- The loop that we're creating is changing a variable outside of itself. And that can make it difficult to reason logically about what the code is doing.
- When we create an instance of this object, we have to use the new keyword. If we forget to use the new keyword, our code is going to fail, pretty silently. And as a result, these kinds of errors can be easy to make and difficult to find, when you're working with Object Oriented code.
- The code is still fairly brittle. It's not very portable. And I would argue that, although this code is easier to test than the earlier code we created, it's still more difficult to test than it needs to be, because our object definition is spread across the prototype in multiple places.

As JavaScript has become more complex and the requirements on both the front end and the back end has been introduced, the importance of testing has come into play. That's one of the reasons why people have started to turn to JavaScript's functional capabilities, and I'm about to show you how that works.