**Learning Node**

**Objects**

As with the page for functions, much of the material on this page is class JavaScript, but some examples are particularly common in Node.js

A lot of this material is review – especially the material on *this* and closures – but since both will used extensively in Node.js this is a fine time to review it.

The material in 2e and 2f is new, so please pay special attention to it.

The key items to review are:

1. **The object hierarchy**
2. **Creating objects**
3. **This and that (or self)**
4. **Prototypical inheritance**
5. **Using the prototype**
6. **The object hierarchy**There is a global object and you may instantiate it with

myObj = new Object();

There are also built-in Objects, such as Array, Date, etc.
These are objects which come from the *JavaScript language.*

In JavaScript it is customary to capitalize the first letter in the object name.

Every object comes with a prototype, which is essentially a template for all objects created with a given constructor function.

You may use that prototype to change or add to the properties and methods of all such objects. (Strongly not recommended for built –in objects.)

As we will see below, prototypes are also used for inheritance, and so build an object hierarchy.
NOTE: There is a separate hierarchy of objects for the *navigator* or browser.

We will not be concerned with those here, but a good diagram of those may be found at <http://www.comptechdoc.org/independent/web/cgi/javamanual/javaobjheir.html>
This hierarchy become important when you are trying to maninpulate the DOM, but not in our discussion of JavaScript (language) objects.

1. **Creating objects**
	1. **A one-off approach or an object initializer**

It is possible to create a one-of-a-kind object (which is of type Object), but this is of limited usefulness:

var myID = {name: “Margaret S. Menzin”,

 email: “menzin@simmons.edu”
 }

This example makes it clear that *objects are implemented as associative arrays.*

* 1. **Using a constructor and this**

If you expect to have more than one instantiation of a type of object then you should define a constructor function.
For example, in paragraph 6 of the Functions page, we first defined a Point() constructor and then instantiated two points:

 function Point(x\_value, y\_value) {
 this.x = x\_value;
 this.y=y\_value;

 this.show = function() {

 alert(“( “ + this.x + “, “ + this.y + “ )” );

 }
 this. reflectAboutXAxis = function(){

 this.y = -this.y;

 }
 }

 origin = new Point(0,0);
 upperRight = new Point(10, 5);

 <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects> provides additional basic information about objects.

* 1. **Cloning an object from another object with Object.create()**

Suppose that I want a new object ColoredPoint. I can accomplish that with:

function ColoredPoint(col, theX, theY) {

 Point.call(this) //This line calls the Point constructor assigning
 //the **properties** to this object – i.e. to
 //ColoredPoint

 this.color = col; //We add a new property to ColoredPoint
 this.x = theX; //Now set the properties inherited from Point

 this.y = theY

}

ColoredPoint.prototype = Object.create(Point.prototype);

 //This makes ColoredPoint’s prototype inherit
 //the **methods** of Point’s prototype.
 //This is the **best practices** way to extend an
 //object (Point) to another object (ColoredPoint)
blueOrigin = new ColoredPoint(“blue”, 0, 0);

blueOrigin.show() //show method was inherited from Point

console.log(blueOrigin.color); //new properties are also available

See <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model> for more examples and
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create> for details on Object.create()

* 1. **Assigning a prototype from another object**

 function ColPoint(col, theX, they) {
 this.color = col;

 }  ColoredPoint.protoype = new Point():

 Now, for an instance of ColPoint you also need to set the properties individually.

 See <http://helephant.com/2009/01/18/javascript-object-prototype/> for more examples

* 1. **Using the prototype versus using this/the constructor – the following material is subtle:**
		1. When you define a method inside the constructor – as was done with show() and reflectAboutXAxis() – are add it to the constructor – as may be done by coding
		*outside*  the constructor:

		Point.distance = function() {
		 d1 = this.x\*this.x + this.y\*this.y;

 return Math.sqrt(d1);

 };
so , you are adding to the constructor, *each instance of Point will have its own show(), distance(), etc. methods.*

* + 1. When you add via the *prototype* by coding

		 Point.prototype.distance = function() {
		 d1 = this.x\*this.x + this.y\*this.y;

 return Math.sqrt(d1);

 };
*then there is one distance function which will be shared by all instances of Point and its extensions* (e.g. ColoredPoint).

* + 1. *While both these methods work*, there are subtle consequences to your choice – and best practices is *usually* to use the prototype.

		In general, the prototype method is more space efficient (and may or may not be more time efficient), but there will usually not be
		big differences either way. That said, here are some good rules to follow:

 1. It is more efficient (& best practices) to put methods on the
 prototype than in the constructor, but for baby examples it won't
 matter.
 And you will see lots of examples where the methods are defined
 with this.
 In other words, while using the prototype is recommended, both
 approaches are common and acceptable.
 That said, there are some situations in which one approach is
 mandated – see next rules.

 2. If your object is something which you will extend, then the methods
 belong on the prototype.

1. If the method is something which you will use as a closure (i.e. it returns a method or an array of methods), then do NOT put it on the prototype.
2. Unless you want all the instances to share it, an array or object you should never put an array or object on the prototype.

 All this is explained in detail at the following references:

* <http://stackoverflow.com/questions/422476/setting-methods-through-prototype-object-or-in-constructor-difference?lq=1>  on the differences between the two approaches.
* <http://stackoverflow.com/questions/6276175/adding-methods-to-custom-objects-in-javascript?lq=1>  on the pros and cons of each approach.
* <http://stackoverflow.com/questions/310870/use-of-prototype-vs-this-in-javascript>  on subtleties when the functin has a method of its own.

**f. In Node**In Node there is a module named util (for now think of a module as a library) with an **inherits** method which helps us.

 var util = require(‘util’); //This imports the module, assuming you have
 //installed it.

 function ColoredPoint(col, theX, theY) {

 Point.call(this)(theX, theY);
 //This line calls the Point
 //constructor, setting up the properties

 this.x = theX;
 //These two lines assign the properties
 //of Point to this object – i.e. to ColoredPoint

 this.y = theY;

 this.color = col; //We add a new property to ColoredPoint

}

util.inherits(ColoredPoint, Point);

 //This takes care of making ColoredPoint’s prototype point to
 //that of Point – i.e. getting ColoredPoint to inherit
 // the methods of Point

In Node many objects extend (or inherit from ) the important **EventEmmiter** object.
In the constructor for such an object you will find code like *EventEmmiter.call(this)*

1. **In ES6** you can accomplish this with classes, but class is really just syntactic sugar for those who still think in java.

1. **this and that (or self)**The constructor function return the value of the **this** pointer – i.e. the newly created object.
When we use Point.call(this) in the ColoredPoint constructor, this refers to ColoredPoint and the call method is asking that the properties be added to ColoredPoint.

There are many errors which programmers make in the use of ***this*,** especially with regards to closures.

I think that the clearest explanation is at <http://www.w3schools.com/js/js_function_invocation.asp>
Here w3schools makes the important and often misunderstood points that **a function which is invoked without belonging to a specified object will belong to the global object and *this* will refer to the global object.**A more detailed explanation may be found at <https://github.com/getify/You-Dont-Know-JS> in the book on *this and Object Prototypes.* (You may need to read the book on *Scope and Closures*  first – but they are short and well worth your time.)

Finally, the context for ***this*** may be specified with JavaScript’s **call, apply, and bind methods.** .)

A simple, clear explanation for call is at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call> in the paragraph “Using call to invoke a function and specifying the context for *this”.*

### Using call to invoke a function and specifying the context for 'this'

In below example, when we will call greet the value of this will be bound to object i.

function greet() {

 var reply = [this.person, 'Is An Awesome', this.role].join(' ');

 console.log(reply);

}

var i = {

 person: 'Douglas Crockford', role: 'Javascript Developer'

};

greet.call(i); // Douglas Crockford Is An Awesome Javascript
 // Developer

Further simple examples are at <https://codeplanet.io/javascript-apply-vs-call-vs-bind/> and near the bottom of <http://www.w3schools.com/js/js_function_invocation.asp>

1. **Prototypical inheritance**As you have seen in the ColoredPoint example, we can make one object extend (inherit from ) another. And we can make the chain as long as we wish.
For instance, at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model> there is an example of a chain with multiple levels.

If we have objects Poodle, Dog, Pet (with the obvious inheritance), when JavaScript is accessing a method or property, it will look first in Poodle (& its prototype) ; if the identifier isn’t there, then JavaScript will follow the prototype chain down to that of Dog, and look for a binding there. If the identifier has not yet been found, JavaScript will continue on to the prototype of Dog.
This is called prototypical inheritance or prototype chaining.

Obviously, if Pet has a method talk, we may over-ride that with a talk method in Dog. Assuming that fluffy is an instance of Poodle, and that Poodle hasn’t overwritten the talk method of Dog, invoking fluffy.talk() will call the talk method of Dog.
2. **Using the prototype**We can add methods and properties to an object or to one instance of an object. Using the example directly above:

 fluffy.prototype.sire =””;

will add the sire property to fluffy (and make it the empty string).

 Poodle.prototype.sire = “”;

will add the sire property to all Poodles, including those such as fluffy which have already been instantiated.

Dog.prototype.sire =””;

will add the sire property to all Dogs, including Poodles and any Poodles such as fluffy which have already been instantiated.

 Pet.prototype.sire = “”;

will add the sire property to all Pets, and so to all Dogs and so to all Poodles and so to fluffy.

Understanding how prototypical inheritance works (see previous paragraph) should make it clear why JavaScript has this surprising feature: you can modify objects and their instances which have been previously defined and used.