**Learning Node**

**Functions, Functions and More Functions**

All the material on this page is review.

The Udemy course **Learn and Understand Node.js with Anthony Alicea** is really useful.

Not only will you *understand* what is going on “under the hood”, but by virtue of spending a lot of time reading his code you will become very comfortable with some of the more sophisticated uses of functions.

Drawing on multiple sources (including my own CS321 material) I have put together the key features of functions.

Most of the material on this page is classic JavaScript, but some of the features listed here are less common in front-end coding and used heavily in Node.

To summarize, here are the key ways that functions are written and used:

1. Plain vanilla functions
2. Function expressions aka function literals
3. Anonymous functions
4. “Functions are first class objects” in JavaScript
5. Functions as methods in objects
6. Constructor functions
7. Functions (or properties) added to prototypes

Note: Constructors & prototypical inheritance is on the objects page

1. Callback functions
2. Closures
3. Immediately invokable functions
4. Function (and variable) definitions are hoisted.
5. Arrow functions.

**So here we go:**

1. **Plain vanilla functions**

We’ve all written this:

function myFunc(param1, param2) {  
 //some code

}

*Note*: Best practices is to open the { as shown on the line with “function “ and to use semi-colons at the end of each line of code  
  
Examples:  
*For a web page:*

function sayHW().{

alert(“Hello World”);

}

*Inside a .js file to be run by node:*

function sayHW().{

console.log(“Hello World”);

}

*For either a web page or in node:*  
function double(x) {

return 2\*x;  
}

1. **Function expressions aka function literals**

We can define a function “on the fly” and assign it to a variable:  
  
 var cubed = function qub(x) {

y=x\*x\*x;

return y;

}

In this case we invoke the function as :

b = cubed(3);

1. **Anonymous functions**

If you plan to use your function recursively, then you need to be able to call qub (example directly above) from within the function,

but if not then the much more common way to use a function expression is to make the function *anonymous:*var cubed = function(x) {

y=x\*x\*x;

return y;

}

And we still invoke it as above.  
This takes some getting used to but the more code you see like this the easier it becomes.

There will be lots of examples on this page.

1. **“Functions are first class objects” in JavaScript**

Well, how could we write about functions in JavaScript without using this phrase.  
What it means is that a function may be used the way any other object is used – it may be a parameter to another function, or

assigned as a property to a method, or the return value of another function.  
  
It also means that any function is an object – and so implemented as an associative array. It also means that any function may have

its own properties and methods (e.g. helper functions).

* 1. **Example of a function as a parameter**:  
     The sort() method for arrays has an optional parameter which is the function to be used for comparison.

(This is needed to get true alphabetic or numerical sorting, as opposed to using the ASCII collating sequence).  
A simple example from <https://msdn.microsoft.com/en-us/library/4b4fbfhk(v=vs.94).aspx> is

var a = new Array(4, 11, 2, 10, 3, 1);

var b = a.sort();

document.write(b);

document.write("<br/>");

// This is ASCII character order.

// Output: 1,10,11,2,3,4)

// Sort the array elements with a function that compares array elements.

**b = a.sort(CompareForSort);**

document.write(b);

document.write("<br/>");

// Output: 1,2,3,4,10,11.

// Sorts array elements in ascending order numerically.

function CompareForSort(first, second)

{

if (first == second)

return 0;

if (first < second)

return -1;

else

return 1;

}

Here the CompareForSort function is a parameter to the sort() method.  
  
 BTW in some situations we actually write the function expression as the parameter!

This is done often for **callback functions (see below).** Example**:** function foo(x, y, bar) {

//do something with x and y, generating variables u and v  
 bar(u,v) //call the bar() function which was passed in.

}

You can then call foo with different values for x, y, and bar().

* 1. **Example of a function which returns a function:**

function multiples(x) {

return function(y) {

return x\*y

}

}

var d = multiples(2); // this is a function which multiplies everything by 2 ---e.g. d(10) is 20.

You could also ask for multiples(2)(10);

1. **Functions as methods in objects**

Because “a function is a first class object” we may assign it to a key in an object.  
Example:

function sayHi() {alert( “Hello world”)}

var day1 = {

english:”Monday ”,   
 french: “lundi ”,

friendly: sayHi

}

The method *friendly* has been assigned the *sayHi* function as its value.

We can get ahold of it as  
 day1.friendly  
 and when we want to invoke it we write

day1.friendly();   
Invoking the function will cause the alert to appear.

Please notice that there are NO parens when we assign sayHi to friendly, but we NEED the parentheses when we want to invoke the function.

We could, of course define our method using an anonymous function, as is done for sympatico:  
  
var day1 = {

english:”Monday ”,   
 french: “lundi ”,

friendly: sayHi,

sympatico: function() {

alert(‘Ola’):

}

}

The sympatico method works just like the sayHi method:

day1.sympatico();

At the risk of stating the obvious, the named method (as is used for friendly) works better if sayHi is going to be used

in other places or called recursively; the anonymous method (as is used for sympatico) is useful if this the only place

where that code will be used.  
  
We can also add methods to a previously defined object in the same way that we added properties to a previously defined object..

Example: Suppose that grades is an array of numbers and we wish to add a method which scales the grades.

grades = new Array(90,80, 70, 85, 92);

grades.course = “CS 321”; //add the course property to grades

grades.scale = function () { //add the scale method to grades.

sum =0

for (i=0; i<grades.length; i++) sum+=grades[i];

av = sum/grades.length;

fudge = Max.max(0, 85-av);

for (i=0; i<grades.length; i++) grades[i]+=fudge;

}

alert(grades.course); //alerts CS 321

grades.scale(); //Invokes the scale method on the numbers stored in the grades array.

The code for the grades.scale method might also be written using this, which, of course, will point to the grades object.

In that case, the code will look like:

grades.scale = function () { //add the scale method to grades, using an anonymous function expression

sum =0

for (i=0; i<this.length; i++) sum+=this[i];

av = sum/this.length;

fudge = Max.max(0, 85-av);

for (i=0; i<this.length; i++) this[i]+=fudge;

}

Please notice that when a function is assigned to a key inside an object (as in the day1 example) we use a colon,

but when a function is assigned to a key or a variable in a stand-along statement (as in athe cubed and grades.scale examples)

we use the = for assignment.

1. **Constructor functions**

When a function is used to instantiate new objects with the certain properties and methods we call it a *constructor*.

A constructor is just a function which, when called with the *new* operator, returns an object (actually a pointer to the object)

-see <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain> for complex examples.  
  
Any object has a property named **this** which points to itself.  
  
First we define the function:  
  
 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;

}   
 }

We may now create new Points:  
  
 origin = new Point(0,0);  
 upperRight = new Point(10, 5);

When we ask for a new Point the Point function will return *this*, a pointer to the just created object.

So we may assign the value of that pointer to origin or upperRight, etc.

And test our methods:

upperRight.show();

upperRight.reflectAboutXAxis();

upperRight.show();

By default the constructor will return the newly created object – you may over-ride that default behavior with a return statement.

1. **Functions (or properties) added to prototypes (of objects created with constructors)**

Every object has a prototype and a property named **prototype** which points to that object.

We can add properties and methods to that protoype after the object has been defined.

Example: Suppose that Point is defined as above.

Point.prototype.distance = function(){

d1 = this.x\*this.x + this.y\*this.y;

return Math.sqrt(d1);

}

By using the prototype we have added the distance method to *all Point* *objects – including those*

*which have already been instantiated.*  
Of course, we may also use the prototype to add properties.

More information about constructors and prototypical inheritance will be found on the Objects page.

1. **Callback functions**

In general a callback function cb() is a function which is passed to some other function foo, with the assumption

that at some point foo will invoke cb.  
  
They are use extensivly in AJAX and in jQuery.

For a jQuery example (see  
<http://www.w3schools.com/jquery/jquery_callback.asp> ) in typical code

$(*selector*).hide(*speed,callback*);

we would pass a selector, a speed (slow or fast) and a function (the callback) to be executed when the hiding is done

e.g. $(‘p’).hide(‘slow’, function(){alert(‘all gone’);} )

The ‘all gone’ alert would appear after each paragraph was hidden.  
  
In this example, hide is defined in jQuery so as to invoke the function in the 2nd parameter when it completes hiding the selected items.

And we have defined the callback with an anonymous function.

We could, however, have put a named function in the 2nd parameter.

We could also have defined our own function foo which executes a callback as follows:  
  
 function foo(param1, param2, callbackFunc) {

//do some stuff and set vars var3, var4, var5

callbackFunc(~~vars~~ var3, var4, var5); //here we are explicitly creating the call back!

//the callbackFunc doesn’t have to have a parameter, or it

//could use one or more of those passed to foo.

};

function bar (param3, param4, param5) { //this is what we will pass to foo

return param3\* (param4 + param5)

};

foo(5, 7, bar); //we invoke foo passing it values for param1 and param2 and bar as the function to be called back.

See <http://recurial.com/programming/understanding-callback-functions-in-javascript/> for more examples.

1. **Closures**

In order to understand how closures work we should first review nested functions (and scope of variables) and how functions can return other functions.  
  
Before starting this section you should read <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.**

* 1. **Nested functions and scope of variables**

In JavaScript it is perfectly possible to have nested functions. For example, if we have code:  
//Some initial code

var globalVar1 = 7

function outerFunc1(param1) {

var outerVar1 = 17;

//some code

function innerFunc1 () {

var innerVar 1 = 13;

someVar;

//some code

} //end of innerFunc1

function innerFunc2 () {

var innerVar 2 =2 3;

//some code

} //end of innerFunc2

//some more code for outerFunc1, including calling innerFunc1().

} //end of outerFunc1

function outerFunc2() {

var outerVar2 = 27;

//some code for outerFunc2

} //end of outerFunc2

//back to main part of script

Key points:

* *Any variable defined inside a function and preceeded with the word var is local to that function* someVar is a global variable b/c, even tho’ it is defined inside a function, it is not preceeded with the word var.  
   If someVar had had its value set inside innerFunc2, it would be immediately avaialble throughout the code – this is  
   because JavaScript hoists functions and variables to the top of the code.

*Note: Trying to log a variable whose value is undefined will throw an error (& prevent or terminate execution) .*

* *Inside a function you have access to the variables in the calling environment and your own variables –i.e.* outerFunc1 has access to the global variables (here globalVar1 and someVar once its value is set) and its own variables (here outerVar1).   
   innerFunc1 has access to the global variables, the variables of outerFunc1 and its own variables (here  
   innerVar1 and someVar, which is a global variable b/c there is no var in its declaration.  
   innerFunc2 has access to the global variables, the variables of outerFunc1 and its own variables (here innerVar2)  
   outerFunc2 has access to the global variables (here globalVar1) and its own variables (here outerVar2).  
   the main body of the script has access only to the globalVar1 and someVar (once its value has been set.)
* *Each time a function is called an execution record is put on the stack, with all its variables.* *Access to the local  
  variables ends (except for closures) when the function is done executing.*
* *Naming conflicts are handled by starting at the top of the stack and working down until the variable is found.*
* *A function may call a function defined immediately inside itself (not 2 levels in) and at its own level –i.e.* outerFunc1 may call outerFunc2 and innerFunc1 and innerFunc2  
   innerFunc1 and innerFunc2 may call each other  
   outerFunc2 may call outerFunc1.  
  1. **Functions which return functions**

Because functions are first class objects, they may be a return value for a function.

In paragraph 4b we saw such an example.

function multiples(x) {

return function(y) {

return x\*y

}

}

var d = multiples(2); // this is a function which multiplies everything by 2 ---e.g. d(10) is 20.

Here multiplies is the outer function and the inner function (which is anonymous) is the one which is returned.

The inner function makes use of the value of x, which was passed to the outer function as a parameter.

For fun, you can test your knowledge at <https://www.youtube.com/watch?v=hRJrp17WnOE>

* 1. **Closures – including the classic closure conundrum (& its solution)**When a function is done executing, its (local variables) are no longer available --- but what happens when  
     that function returns another function whose execution depends on the values of the outer function’s   
     local varaibles? Answer: *The returned (i.e. the inner) function still has access to the environment which   
     created it, and so to the local variables of the outer function! This is called a closure.*  
     For an absolutely clear example see <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>   
     (up to the part about emulating private methods).  
       
     The classic closure conundrum arises when you mix together closures and loops. The reference above has one

example of what can go wrong. Here is another, which causes problems even without returning user-defined functions.  
  
function powersTo(n) {  
 powerful = new Array(n+1); //Has n+1 entries indexed from 0 to n

for (j = 1; j < n+1; j++) {

powerful[j] = function(x) { return Math.pow(x, j); } //powerful is an array of functions

}

return powerful;

}

pw = powersTo(3);

for (k = 0; k < 4; k++) {

console.log( pw[k](10) );

}

*The result is 1000 1000 1000 1000 --- because when the powersTo function finished executing and returned powerful,*

*the value of j was its value at the end of the for-loop in powersTo, and so all the functions in pw return that power of 10.  
  
Solutions*: (1) Use ES6 and the let statement . The new let statement defines a block scope variable.

// see <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let> for more information.

let n=5;

let powerful=new Array(n);

for (let j = 0; j < n; j++) {

//Notice that j is a let variable and so we get a new  
 //context for each iteration of the for loop.

powerful[j] = function(x) { return Math.pow(x, j);

}

};

for (let k=0; k<n; k++){

console.log(powerful[k](2));

}

Or (2) create a new environment for each iteration of the for loop in powersTo, so giving each entry in powerful its own value of j.  
(This can also be accomplished by wrapping things in immediately invokable functions.) This is an old-fashioned way to do things, and today method (1) is the standard.

for (j = 1; j < n+1; j++) {

function p(j){

var jHere = j;

powerful[j] = function(x) { return Math.pow(x, jHere);  
 };

p(j);

}

If your closures are not working the way you expected then the best way to understand them is at   
<https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/README.md#you-dont-know-js-scope--closures>   
and especially at <https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/README.md#you-dont-know-js-scope--closures>   
  
  
As Kyle Simpson explains, what matters for a variable or reference is the *context in which it is called.* A common problem is the context for *this* and whether or not you mistakenly have a global context.  
  
and at some of the articles we read in Chapter 5.

1. **Immediately Invokable Functions and Immediately Invokable Function Expressions (IIFEs)**

**Immediately Invokable Functions:** If you want a function defined anonymously to be invoked inmmediately you can do so adding the () to invoke it.  
  
Example:

If we modify the day1 example (paragraph 5) by adding friendlyNow:

function sayHi() {alert( “Hello world”)}

var day1 = {

english:”Monday ”,   
 french: “lundi ”,

friendly: sayHi,

friendlyNow: sayHi**()**

}

then *friendlyNow* will get executed when the code is read and you will *not* be able to be invoked later.

(That is, nothing happens when you write day1.friendlyNow.)

**Immediately Invokable Function Expressions (IIFEs):** If you want a function defined anonymously to be invoked inmmediately you can do so by wrapping it in  
parentheses and adding the () to invoke it.  
  
Example:

**(**function(){alert(‘Goodbye’);**)()**

Notes:

1. Wrapping this function in the **( )** turns it into a *function expression.*  
2. Some people place the **()** inside the **( )**; it means the same thing:   
  
 **(**function(){alert(‘Goodbye’); **())**  
3. You can also have parameters in an IIFE:  
  
 **(**function(name ){alert(‘Goodbye ’) + name;}**)(‘Sally Simmons’)**

1. **Function (and variable) declarations are hoisted**

This means that the declarations of functions and vars are put at the top of the scope they are defined in – with the function declarations above the var declarations. So you may use a function (within the same scope) above where you have typed it on the page.  
  
***Please recall and note that function expressions are not hoisted.  
  
Because arrow functions are function expressions, they are not hoisted.***  
A further discussion of this may be found at <https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20%26%20closures/ch4.md>

1. **Arrow Functions**These were discussed extensively in the Class-by-Class Assignments for Chapter 5, and the various format are summarized in the document Arrow Function Syntax, in Chapter 5 <http://web.simmons.edu/~menzin/CS321/Unit_2_JavaScript_and_HTML_Forms/Chapter_5_HTML5_Innovations/>. To reiterate, we had a summary table:  
   **Summary table – commented version**

|  |  |  |
| --- | --- | --- |
|  | **One parameter** You may omit () around that solo parameter | **Not one parameter** Parentheses are required for no parameters or for more than one parameter |
| **Body is one expression** No { ] and return is implicit. | x => some\_expression \_in\_x; | (x, y) => some\_expression\_in\_x\_and\_y; () => some\_expression; |
| **Longer function body** | x=> {//some code  //end with return value  } | (x,y) => {//some code  //end with return value  } () => {//some code  //end with a return value  } |

**Summary table – uncommented version**

|  |  |  |
| --- | --- | --- |
|  | **One parameter** | **Not one parameter** |
| **Body is one expression** | x => some\_expression \_in\_x; | (x, y) => some\_expression\_in\_x\_and\_y; () => some\_expression; |
| **Longer function body** | x=> {//some code  //end with return value  } | (x,y) => {//some code  //end with return value  } () => {//some code  //end with a return value  } |