**Week 4 Notes** – JavaScript and the DOM

**PASSING AND RETURNING FUNCTIONS**

Passing Functions

Functions can call other functions

forEach is built into JS as a method for array objects ( arr.forEach(function) )

Returning Functions

*example*

function square(x) { return Math.pow(x,2) };

*similar to above*

function toTheN(n){

return function(x) {

return Math.pow(x,n);

};

}

var square = toTheN(2);

var toThe3rd = toTheN(3);

document.getElementById("out1").textContent = square(2);

document.getElementById("out2").textContent = square(3);

document.getElementById("out3").textContent = toThe3rd(3);

In this case, square returns function(x), which itself returns Math.pow(x, n), when you call Square(2), you are returning Math.pow(2, 2) because the n in Math.pow(x, n) was defined in the assignment

var square = toTheN(2)

Higher Order Functions

Functions that take in a function as an argument or potentially return a function as a result (not formal definition, close enough for this class)

callbackhell.com – examples of messy function definitions that result in very ugly code

Note: Web development JS uses largely asynchronous programming, i.e. where multiple things are happening at the same time, so the majority of the programs will be functions being passed by other functions

**SCOPE AND CONTEXT IN JAVASCRIPT**

Scope: where a variable can be accessed from

Context: describes where a method has been called from

Execution Context

this keyword allows you to access the properties of an object instance when you are invoking the method of an object instance

“this” specifies who is responsible for calling a function

console is a global object, but passing console.log to a higher-order function gives a new execution context of the new function. Before, when called via “console.log” the execution context was the “console” object, so inside that function, “this” refers to the console. But “this” now refers to the *functions* execution context, so if at some point in “console.log” something like this.someConsoleProperty is used, it will fail because the “forEach” context does not know about “someConsoleProperty.” The “console.log” function does not know that it is no longer in the console context‼

*‘bind’ Function*

bind is a method of a function (remember: functions count as objects in JavaScript) that lets us specify an execution context for a function, and can let us fix an arbitrary number of the first args passed to the function.

e.g

func.bind(thisArg[, arg1[, arg2[, ...]]])

thisArg – provides execution context for the new function

arg1, arg2, argN – arguments to prepend to the list of arguments passed to the function, allows you to fix certain args so that they are always passed to the resulting function

return – returns the *new function*, does not change the function but returns the new one

now, for the console problem fix:

function forEach(a, work) {

for (var i = 0; i < a.length; i++) {

work(a[i]);

}

}

var arr = [1, 2, 3];

//forEach(arr,console.log); <--- This will not work.

var newLog = console.log.bind(console); //newLog now has the console.log function with a fixed context of console.

forEach(arr, newLog);

Now you can make this work!

Scope

JavaScript *only* has function scope and global scope

When a function is defined inside of another function, there are some oddities, and can result in bugs

function outer(){

var n = 0;

inner();

return n;

}

function inner(){

n += 1; //Throws an error, n is not in scope.

}

//outer();

function closureMaker(){

var n = 0;

var inner = function(){

n +=1;

};

var outer = function(){

inner();

return n;

};

return outer;

}

var outerClosure = closureMaker();

console.log("Call 1: " + outerClosure()); //Call 1: 1

console.log("Call 2: " + outerClosure()); //Call 2: 2

Looping can have some unwanted results within these rules e.g.

function generateFunctionList(){

var fnArr = [];

for( var i = 0; i < 3; i++){

fnArr[i] = function(){

console.log(i);

};

}

return fnArr;

ABOVE always returns 3! *i* is never incremented past 3 and is always called, to resolve, we need to give each function its own copy of the value, so:

function generateFunctionList(){

var fnArr = [];

for( var i = 0; i < 3; i++){

fnArr[i] = function(x){

return function(){

console.log(x);

};

}(i);

}

return fnArr;

Remember: only *ONE* closure is generated per call.

From video I found:

<https://www.youtube.com/watch?v=71AtaJpJHw0>

JavaScript will look outside of the scope of its function for a variable until it finds it or not. If it finds that value, it adds it to the scope of the function

“Closures are functions with preserved data,” a record storing a function together with an environment

**OBJECTS AND CLASSES IN JS**

Constructors

ex: DVD class

function Dvd(title, length, rating){

this.title = title;

this.length = length;

this.rating = rating;

this.watchCount = 0;

}

Calling this function by prefixing it with the “new” keyword will give back an instance of the DVD class, e.g.

var myDvd = new Dvd("Ocean's Eleven", 117, "PG-13");

Difference between C++ and JavaScript → JS classes use a prototype. If you call a method or request a property of an instance of an object and it can’t be found the JS will look at the prototype object to see if the property exists there. If you have something that is shared by *all* instances of a class, it should be a prototype. Can add like so:

Dvd.prototype.watch = function(){

this.watchCount++;

console.log("You watched " + this.title + ". That is a good movie!");

};

You wouldn’t want watchCount on the prototype because every DVD would share the same watch count that is not unique to an individual DVD

The Function’s ‘call’ Method

‘call’ is in the same neighborhood as ‘bind.’ It is a method of a function, but instead of creating a function copy, it calls it immediately within a specific execution context

ex:

function timeToWatch(freeTime){

return freeTime > this.length;

}

And calling that:

timeToWatch.call(myDvd, 200);

This sets the context to ‘myDvd’

*Another example of constructors and prototypes:*

Function Student(name) {

this.name = name;

}

Student.prototype.sayName = function() { console.log(this.name);

};