Lecture 5

What we discussed in the last class

- **What is meant by scope?**

Scope defines the memory (variables) that can be accessed in a code block. For e.g.

```
#include <stdio.h>

int main(void) {
        int a = 10;
        printf("%d,", a); // First print
        int i;
        for(i = 0; i < 3; i++) {
            int a = 15;
            printf("%d,", a); // Second print
        }
        return 0;
}

Output : 10,15,15,15
```

In this case the print command would print the inner variable value i.e. **10** in the first print command and then throughout the loop print **15**. This is showing static scoping or **lexical scoping**.

Lexical scoping means that the variables in the block nearest to them would get priority over the variables in blocks further off in the nesting.

Blocks are created using braces in c like languages, so the main function creates a block, for creates a block inside the main block.

On the other hand there is something known as **dynamic scoping.**

```
myVar=0;

function foo(){
    var myVar = 10;
    bar();
}

function bar(){
   console.log(myVar);
}

foo();
bar();
```

Dynamic scoping means that the scope given to a function call depends on where it is invoked from from.

The above code snippet in static scoping should print **0** for both the calls but if dynamic scoping is present then foo call would print **10** and bar call **0.**

This is because in the function foo, bar is called and given the local context on the function foo.

More clearly explained here :
https://msujaws.wordpress.com/2011/05/03/static-vs-dynamic-scoping/

Javascript Specific Scope

Javascript luckily follows lexical scope, that too only ***functional lexical scope***. What this means is that the only thing in javascript that can separate scope is a function. Writing a code similar to the first one in javascript should throw errors because of defining the variable twice in the same scope.

Hoisting in Javascript
All variables and function definitions are internally made to reach the top of the function definition in javascript, so don't rely on where you define the variables in the function ever, because internally they are going to reach the top of the function. This is known as **hoisting in javascript**.

Javascript follows functional scope everywhere except for the **this** keyword, I would be talking about **this** in next class. You can read about it here:

https://spin.atomicobject.com/2014/10/20/javascript-scope-closures/

- ## Arrays and Objects.

As already discussed the right answer to every question in javascript is objects. In javascript everything (except for a few things) is an object.

Arrays

var a = ['apples', 'oranges'] is an array defined having 2 values apples and oranges. As array is an object this would have its keys, the keys here are 0 and 1 (the indexes of array).

Internal representation of this array would look something like this,
```
var a = {
       0 : 'apples'
       1 : 'oranges'
}
```

Objects

When you create a new object you are essentially extending the **prototype chain.** Think of the prototype chain as a normal worldly chain. If you are searching for a property for an object and you don’t find it, you go one level up in the prototype chain to find the property till you encounter null.

```
// Let's assume we have object o, with its own properties a and b:
// {a: 1, b: 2}
// o.[[Prototype]] has properties b and c:
// {b: 3, c: 4}
// Finally, o.[[Prototype]].[[Prototype]] is null.
// This is the end of the prototype chain, as null,
// by definition, has no [[Prototype]].
// Thus, the full prototype chain looks like:
// {a: 1, b: 2} ---> {b: 3, c: 4} ---> null

console.log(o.a); // 1
// Is there an 'a' own property on o? Yes, and its value is 1.

console.log(o.b); // 2
// Is there a 'b' own property on o? Yes, and its value is 2.
// The prototype also has a 'b' property, but it's not visited.
// This is called "property shadowing."

console.log(o.c); // 4
// Is there a 'c' own property on o? No, check its prototype.
// Is there a 'c' own property on o.[[Prototype]]? Yes, its value is 4.

console.log(o.d); // undefined
// Is there a 'd' own property on o? No, check its prototype.
// Is there a 'd' own property on o.[[Prototype]]? No, check its prototype.
// o.[[Prototype]].[[Prototype]] is null, stop searching,
// no property found, return undefined.
```

**Code taken from*
https://developer.mozilla.org/en/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

The above example very well illustrates the concept of prototype chains, so how is an object created or more succinctly extended from the prototype chain?

There are 3 methods,

1st method:

```
var o = { a : 1 } // This is shorthand notation to create an object
```

2nd Method and more complete method is :

```
function foo() {
        this.someValue = 0;
}

foo.prototype = {
        increment : function() {
                this.someValue++;
        }
};

var objectInstance = new foo();
```

Foo method in the above example is not the class, there are no classes in javascript. It is just the **constructor**. A constructor's job is to create the object, initialize the properties.

In javascript’s case the constructors are simply functions which when called with the **new** keyword, extend the object prototype chain.

Note : You can print the prototype chain by doing object.__proto__.__proto__ and so on.

So here the prototype chain for objectInstance looks like this,

{ someValue : 0 } → { increment : function } → Object Prototype → null

The first one is the object itself, the second in the chain is the object prototype containing increment function which then inherits from object prototype and finally ending at null.

The third method is using ES6 notation, now ES6 notation despite using classes is still using method 2 only in the background, do not get confused with the classes.

More about this new notation can be read here :
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

- **Closure**

```
function valueDef(n) {
        var def = n;
        return function() { return def }
}

var x = valueDef(1);
var y = valueDef(2);

console.log(x());
console.log(y());
```

The above code if written will print 1 and 2 respectively.

The function valueDef returns a function, so x and y variables are storing the functions returned by valueDef.

Now in c like languages once function run is over all local variables get removed from memory. If this had happened in javascript the above example should return undefined.

Hence somehow the functions returned by valueDef are still storing the value of def somewhere and is not losing access.

Whenever a function is stored in javascript, it will be stored along with its **function scope.** This functional scope is known as closure.

I will explain this concept once more in class today.

- **Map and Reduce**

Map

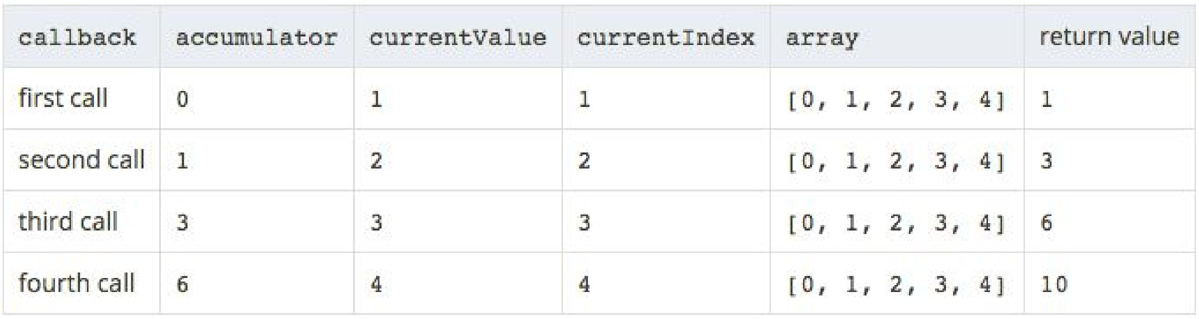
Map function is analogous to the 'functions in maths'. Map takes an array and runs a function on each element and returns a new array.

```
var a = [1,2,3]
var b = a.map( function(num) {
        return num * num;
});

console.log(b);

//prints [1,4,9]
```

More detailed information here :
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Reduce
Reduce function is used to accumulate the values of an array into a single object. For example adding all the numbers together in an array. For reference see :
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

```
[0, 1, 2, 3, 4].reduce(
  function (
    accumulator,
    currentValue,
    currentIndex,
    array
  ) {
    return accumulator + currentValue;
  }
);
```



callback	accumulator	currentValue	currentIndex	array	return value
first call	0	1	1	[0, 1, 2, 3, 4]	1
second call	1	2	2	[0, 1, 2, 3, 4]	3
third call	3	3	3	[0, 1, 2, 3, 4]	6
fourth call	6	4	4	[0, 1, 2, 3, 4]	10

**Code taken from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce*

Example covered in class was to count all instances of the name in an array :

```
var names = ['Alice', 'Bob', 'Tiff', 'Bruce', 'Alice'];
var countedNames = names.reduce(function (allNames, name) {
  if (name in allNames) {
    allNames[name]++;
  }
  else {
    allNames[name] = 1;
  }
  return allNames;
}, {});

// countedNames is:
// { 'Alice': 2, 'Bob': 1, 'Tiff': 1, 'Bruce': 1 }
```

Quiz in the next class on Basics of Javascript.

Next class snapshot

- Understanding this
- Understanding the prototype chain
- Be on time.

Important resources

- **JS Expert :** http://eloquentjavascript.net/

Keep on reading eloquent javascript (JS Expert Link) whenever you have time.