Intro To JavaScript – Day 5

Scope in JavaScript

Scope refers to the visibility of identifiers in a program; such as variables and functions. Scope in JavaScript works differently than what you are used to in C++/C#/Java. Those languages use *block* scope, JavaScript uses *function* scope.

All of the variables defined within a function are hoisted to the top of the function, making them visible throughout the whole of the function. Consider the following code...

```
function demoScope(count) {
    var counter = 0,
        total = 0;

    for (counter = 0; counter < count; counter++) {
        total += counter;
    }

    if (total % 2 === 0) {
        var even = true;
    }
    else {
        even = false;
    }

    if (even === true) {
        console.log('The total is even: ' + total);
    }
    else {
        console.log('The total is odd: ' + total);
    }
}
demoScope(6);
```

- Notice that '`var even`' is declared inside of an `if` block, but it is still visible outside of that block. This is because of the *function* scope, rather than *block* scope used by JavaScript.

(old news) Due to hoisting of `var` declared variables, it has become accepted practice in JavaScript to define all variables at the top of a function, even before they are used.

With ES6 you can use `let`, which uses block scope, rather than function scope.

Functions as Modules (or Namespaces)

It is often useful to place all of the code inside of a well-controlled module (or namespace) so that you can be assured there are no naming collisions with anything else. This is done using what has become known as the Module pattern. Let's illustrate this by creating a new module that performs some additional random number generation capabilities...

```
let Random = (function() {
      'use strict';

      function nextDouble() {
            return Math.random();
      }

      function nextRange(min, max) {
            let range = max - min + 1;
            return Math.floor((Math.random() * range) + min);
      }

      function nextCircleVector() {
            let angle = Math.random() * 2 * Math.PI;
            return {
                  x: Math.cos(angle),
                  y: Math.sin(angle)
            };
      }

      return {
            nextDouble : nextDouble,
            nextRange : nextRange,
            nextCircleVector : nextCircleVector
      };
}());

console.log(Random.nextDouble());
console.log(Random.nextRange(1, 4));
console.log(Random.nextCircleVector());
```

- An anonymous function is defined, immediately executed, and then the return value assigned to a variable (`Random` in this case).
- Notice the return statement creates a new object and defines three properties on it. Those three properties hold the functions defined in the body of the anonymous function. If you wanted, you could name the properties something different. This is actually a closure, with the closure hanging onto those three functions for as long as the returned object lives.
- The `Random` variable goes into the global JavaScript namespace, giving us a name onto which we can place things that won't collide with other functions of the same name.
- It also turns out that `Random` is a singleton, there is only one!

This is how you'll want to organize your game code. Create a single game module, and place everything inside of it.

```
let MazeGame = (function() {

      images: {},

      status: {},

      ...whatever else you want to place in here...

}());
```

With a module defined, it is possible to extend it by adding new modules on it...

```
MazeGame.graphics = (function() {
      ...put rendering code here...
}());
```

Closures

In JavaScript functions are executed using the variable scope that was in effect when they were defined, also known as *lexical scoping* (the same as other languages). Also remember that JavaScript has function scoping, which means that any variables within function scope need to be available when some code is executed. So what, so check out this...

```
function fibonacci() {
    let memory = {0:1, 1:1};

    return function inner(n) {
        if (!(n in memory)) {
            memory[n] = inner(n - 1) + inner(n - 2);
        }
        return memory[n];
    };
}
let myFib = fibonacci();
myFib(10);
```

Yesterday we saw a different approach to the Fibonacci, this is much cleaner and serves to illustrate closures pretty well.

- We define a function named `fibonacci` that returns a function that performs the actual computation.
- Inside of `fibonacci` we define an object named `memory`, used to hold previous fibonacci results.
- The `inner` function checks memory to see if the value exists.
  - If it doesn't, the value is computed (recursively).
  - If it does, the value is returned.

In a language like C++/C#/Java, when `fibonacci` returns, `memory` goes out of scope and its lifetime is over. In JavaScript, this is not necessarily the case. If the object is referred to by another scope that is still alive, that object is kept alive for as long as the other object is alive.

- That means that as long as `myFib` is around, the memory object is kept alive. This is what is known as a closure!

Let's take a look at another example...

```
function counter(start) {
    let next = start;

    return {
        next: function() { return next++; },
        reset: function() { next = start; }
    };
}
let myCounter = counter(10);
console.log(myCounter.next());
console.log(myCounter.next());
myCounter.reset();
console.log(myCounter.next());
console.log(myCounter.next());
```

- This `time` the function returns an object with a couple of functions as properties, we seen this before.
- Inside of counter `next` is defined an initialized with the value of `start`.
- The `next` property/function uses `next`.
- The `reset` property/function uses the parameter `start`

Therefore, both `start` and `next` are kept around for the lifetime of the object returned by the function `counter`, another clever closure!